Object Tracking and Biomimetic Vergence Control for an Active Stereoscopic Vision System

A Project Report Presented to
The Department of Computer Science
Institute of Information and Mathematical Sciences
Massey University, Albany
New Zealand

In Partial Fulfilment of
The Requirements for the Degree of
Bachelor of Science with Honours in Computer Science

Author: Jeremy M. Willemse
Supervisor: Dr. Andre L. Barczak

June 24, 2012
Abstract

This research presents a hardware and software platform for manipulating a servo-based stereoscopic vision system with four degrees of freedom. The system mimics human vision by introducing a vergence control mechanism whereby the two cameras are able to rotate their gaze toward the same point. Geometric properties of the stereo camera system such as the baseline Inter-pupillary Distance (IPD), the Field-of-View (FoV) of the camera lenses and the vergence angle of both cameras are used to determine the next required angles of vergence for tracking a particular Object of Interest (OoI). The Viola-Jones object detection framework is coupled with the camera control software for the purpose of finding and tracking OoIs in the stereo FoV. The result is single binocular vision which has the ability to extract depth information for an arbitrary object of interest with high accuracy, experiments have been conducted to support this.
# Contents

1 Research Description
   1.1 Overview of the Current State of Stereo Vision . . . . . . . . 1
   1.2 Research Objectives . . . . . . . . . . . . . . . . . . . . . 2
      1.2.1 Primary Objective . . . . . . . . . . . . . . . . . . 2
      1.2.2 Breakdown of Objectives . . . . . . . . . . . . . . . 2
   1.3 Scope and Limitations of this Research . . . . . . . . . . . . 2
   1.4 Overview of the Problem Domain and Experimental Platform 3
      1.4.1 Experimentation . . . . . . . . . . . . . . . . . . . 4
   1.5 Significance of the Research . . . . . . . . . . . . . . . . . 4
   1.6 Research Methodology . . . . . . . . . . . . . . . . . . . . 4
   1.7 Structure of this Document . . . . . . . . . . . . . . . . . . 5

2 Review of Related Literature
   2.1 Biological Depth Perception . . . . . . . . . . . . . . . . . 7
      2.1.1 Voluntary Eye Movement . . . . . . . . . . . . . . . 7
      2.1.2 Saccades . . . . . . . . . . . . . . . . . . . . . . . . . 8
      2.1.3 Lens Accommodation . . . . . . . . . . . . . . . . . . 8
      2.1.4 Stereopsis . . . . . . . . . . . . . . . . . . . . . . . . 8
      2.1.5 Non-Stereo Depth Perception . . . . . . . . . . . . . 9
      2.1.6 Field-of-View . . . . . . . . . . . . . . . . . . . . . . 11
      2.1.7 Interpupillary Distance . . . . . . . . . . . . . . . . . 11
   2.2 Computer Stereo Vision . . . . . . . . . . . . . . . . . . . 11
      2.2.1 Epipolar Geometry . . . . . . . . . . . . . . . . . . . 12
      2.2.2 Real Camera Systems . . . . . . . . . . . . . . . . . . 13
      2.2.3 Virtual Camera Systems . . . . . . . . . . . . . . . . 13
      2.2.4 Vergence Control by Fuzzy Logic . . . . . . . . . . . 14
   2.3 Object Detection and Tracking . . . . . . . . . . . . . . . . 18
      2.3.1 Viola-Jones Object Detection Framework . . . . . . . 18
3 Proposed System

3.1 Hardware .................................................. 22
   3.1.1 Design of Hardware ................................. 22
   3.1.2 Fabrication .......................................... 22

3.2 Software .................................................. 27
   3.2.1 Serial Servo Interface .............................. 27
   3.2.2 Camera Interface .................................... 29
   3.2.3 Object Identification and Tracking ................ 30
   3.2.4 Vergence Angle Controller ......................... 30
   3.2.5 Pan and Tilt Control ............................... 32
   3.2.6 Depth Perception and Calculation .................. 34
   3.2.7 Class Structure ..................................... 35

3.3 Summary .................................................. 35

4 Results ...................................................... 37

4.1 Gradual Vergence ......................................... 37
   4.1.1 100 mm Baseline .................................... 38
   4.1.2 250 mm Baseline .................................... 41

4.2 Geometric Vergence ....................................... 43
   4.2.1 100 mm Baseline .................................... 43
   4.2.2 250 mm Baseline .................................... 46

5 Discussion ................................................... 48

5.1 Accuracy .................................................. 48

5.2 Speed ..................................................... 51

6 Conclusions .................................................. 52

6.1 Possible Future Work ..................................... 52

6.2 Conclusions ............................................... 53
# List of Figures

2.1 Binocular Disparity [1] ............................................. 10  
2.2 Epipolar Geometry ................................................. 13  
2.3 Vergence Angles [2] ................................................. 14  
2.4 Stereo Pair from a Reflective Prism [3] ......................... 15  
2.5 Rotated Prism Creating a Pan Effect [3] ......................... 15  
2.6 ZNCC Input Membership Function .............................. 16  
2.7 Depth Input Membership Function .............................. 17  
2.8 Vergence Output Membership Function ......................... 18  
2.9 Rectangular Features used by the Viola-Jones Object Detection Framework ............................................. 19  
3.1 Concept Servo and Camera Design ............................ 23  
3.2 Assembled Servo and Camera Hardware ....................... 24  
3.3 Logitech™ QuickCam™ Pro 9000 ............................... 24  
3.4 Geometric Properties Required for Calculating the Field-of-View ................................................ 25  
3.5 Hextronik HXT900 9 gram Servo ............................... 26  
3.6 Pololu Micro Maestro 6-Servo Micro-Controller ............ 26  
3.7 The USB Dual Port Serial Mode [4] ......................... 27  
3.8 The UML Class Diagram of the Proposed System ............ 36  
4.1 Gradual Vergence (100 mm baseline) X Y Scatter Plot for Five Samples at Eighteen Distances .................. 38  
4.2 Gradual Vergence (100 mm baseline) Mean Values and Error Bars .................................................. 40  
4.3 Gradual Vergence (250 mm baseline) X Y Scatter Plot for Five Samples at Eighteen Distances .................. 42  
4.4 Gradual Vergence (250 mm baseline) Mean Values and Error Bars .................................................. 42  
4.5 Geometric Vergence (100 mm baseline) X Y Scatter Plot for Five Samples at Eighteen Distances .................. 43  
4.6 Geometric Vergence (100 mm baseline) Mean Values and Error Bars .................................................. 45
4.7 Geometric Vergence (250 mm baseline) X Y Scatter Plot for Five Samples at Eighteen Distances . . . . . . . . . . . . . . . 47
4.8 Geometric Vergence (250 mm baseline) Mean Values and Error Bars . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

5.1 Comparative Accuracy Metric Between All Experiments . . . 49
5.2 Delta Vergence Angles Between Two Objects for Two baseline IPDs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
# List of Tables

2.1  Rule Base For the Fuzzy System ........................................... 17
3.1  HXT900 Servo Specifications ............................................. 25
4.1  Gradual Vergence (100 mm baseline) Experimental Results ........ 39
4.2  Gradual Vergence (250 mm baseline) Experimental Results ........ 41
4.3  Geometric Vergence (100 mm baseline) Experimental Results .... 44
4.4  Geometric Vergence (250 mm baseline) Experimental Results .... 46
List of Algorithms

1  AdaBoost Algorithm for the Viola-Jones Object Detection Framework ........................................... 20
2  Geometric Vergence Controller ............................................. 33
3  Pan and Tilt Control ................................................................. 34
4  Calculate Depth ................................................................. 34
Chapter 1

Research Description

1.1 Overview of the Current State of Stereo Vision

Stereoscopic vision is an area of study within the field of computer vision, aimed at deriving three-dimensional scene information from a pair of two-dimensional views. Typically, these two views are obtained using two separate camera sensors, each with a different perspective of the same scene. There are two general methods for extracting depth information using two cameras. The first, the correspondence method, requires the cameras to be positioned with a known relative location to each other, typically this means they are set up with a parallel point-of-gaze. The second, the depth-from-vergence method requires that the two cameras are positioned with a known base distance, however, the point-of-gaze of each camera will be rotate to the location of the object of interest. The two views may also be obtained using a single camera and a series of prisms and mirrors positioned in such a way that both views are projected within the frame of the single camera.

The views are processed in software in order to estimate the distance each object in the scene is from the cameras (i.e., the depth). The techniques aim to replicate stereopsis, the term given to the ability animals and people with normal binocular vision have to see in three-dimensions. This is achieved using the binocular disparity of the object of interest in the two views. In the correspondence method, the disparity is determined by the difference in the observed location of any given point as seen by each camera. Depth-

1More than two cameras may be used, however, within the scope of this research, all stereo vision methods discussed will be limited to two camera views
by-vergence attempts to eliminate the disparity for a particular object of interest, allowing depth for the object to be geometrically solved. This is the method implemented in this work and is also the way in which biological stereo vision functions.

1.2 Research Objectives

This research aims to replicate human stereopsis and binocular vision. Experiments will be conducted to evaluate the accuracy of the proposed system. The individual objectives are described in this section.

1.2.1 Primary Objective

To develop a controller for a servo-based, biomimetic, stereoscopic vision system with four degrees of freedom. The system is to be capable of accurately perceiving depth and tracking a predetermined object of interest such that the gaze of both cameras is directed toward this object through convergence and divergence of two cameras. The vergence angles of each camera will be approximately symmetrical through a pan mechanism. The object of interest should be central within the vertical field of view, using a tilt mechanism.

1.2.2 Breakdown of Objectives

1. Gain experience in the practical process of combining built-for-purpose hardware and software
2. Gain knowledge about the human vision system
3. Gain an understanding of object detection methods
4. Gain in-depth insight to the field of stereo vision and vergence control.
5. Develop a working stereo vision hardware and software system
6. Generate a comparative analysis of depth perception using various configurations

1.3 Scope and Limitations of this Research

The budget limitations of this research require a cost effective means of acquiring a stereo vision head. The most suitable way to achieve this is by
fabricating it using affordable web cameras, servos, and materials. The hardware is discussed in section 3.1. It is expected that this will affect precision and accuracy.

The cameras and object recognition techniques used are only suitable for estimating depth at relatively short distances. No attempt to measure the distance to objects from long range will be made.

The field-of-view of the cameras used is not equivalent to that of the human eye, in this respect, the system is not biomimetic, however, in terms of pan, tilt and vergence functionality, it is.

The nature of depth-from-vergence is that there can only be one object of interest per pair of views. The system does not attempt to determine the depth of objects not located at the point of gaze. Correspondence methods could however be implemented to achieve this.

Depth perception in the system is limited to depth-from-stereo techniques. Humans also gather scene information and depth information from additional means such as light and shading cues. These are not implemented in the proposed system.

The final system is a proof-of-concept and does not reflect exhaustive object detection research. In the present state, object detection does not operate in real-time. As a result, the system suffers some latency delay when attempting to follow a moving object. However, the software is modular and may be further improved.

1.4 Overview of the Problem Domain and Experimental Platform

Although humans do not estimate depth with exact precision, the problem with respect to computer vision is one of numerical accuracy. As such, the various tested configurations of the system will be ranked accordingly.

In order to develop an experimental platform for the proposed system, some method of selecting an object of interest must be established. In section 3.2.3 the Viola-Jones object detection framework is discussed. The face recognition variant of this has been implemented as the object recognition
technique. The system works by fixating the point of gaze of each camera on the detected face.

1.4.1 Experimentation
All experiments performed in this work are undertaken in a laboratory environment. The accuracy of the depth perception methods are measured by using a series of known distances. Multiple experiments are performed in order to determine a margin of error. An attempt to determine the optimal configuration of the system has been made, this involves experimenting with different baseline distances between the two cameras. However, in keeping with the biomimetic theme, a comparative analysis of the optimal configuration and the near-biomimetic baseline configuration has been undertaken.

1.5 Significance of the Research
Although the concept of depth-from-vergence is not new, the related literature as discussed in chapter 2 presents several different methods of converging to a point of interest. Some involving fuzzy logic and similarity measurements, others use mathematical modelling techniques. The system proposed by this research attempts to solve the vergence problem using inverse geometric formulae and lens properties to converge directly to the object of interest in a single command per servo. The result is that fewer serial data packets are required. This method depends a great deal on the angle-of-view of the camera lens. Determining such a value is discussed in section 3.1.2.

1.6 Research Methodology
To achieve the objectives of this research, the following methodology has been followed:

1. Research fundamental theory related to biological stereopsis to develop a model for biomimicry.

2. Research related literature in order to form an understanding of the field and possible approaches.

3. Research object detection methods useful for determining the object of interest
4. Fabricate a low-cost stereo vision system using servos and USB cameras for use as a test platform.

5. Develop a software interface for the servo controller

6. Develop a software interface for gathering information obtained by the camera sensors, including the location of the object of interest

7. Develop a software controller for automating the servo positions based on the camera information

8. Execute experiments to measure the accuracy of various configurations of the system in order to determine a near optimal setting.

1.7 Structure of this Document

The research objectives and methodology which has been followed throughout this work was discussed in this chapter. The scope and limitations have been outlined, the problem domain was analysed and the experimental testing platform was introduced. This includes the camera-servo hardware system and a brief overview of the experimentation processes undertaken. The following chapters in this project report elaborate on these concepts.

Object recognition and binocular convergence are the two fundamental components of the system proposed within this work. In Chapter 2, these fundamentals are abstractly defined in terms which explain the biological model, but may be interpreted by a computational system. Also discussed are several existing computational implementations of the various similar algorithmic approaches used to achieve vergence control for stereo vision systems. A brief introduction to an existing image processing technique that will be implemented in this work is also covered.

In chapter 3, the proposed system developed for this work is introduced. The fabrication of the testing platform hardware is discussed and the properties of each individual component are reviewed. The software features are presented in an easy-to-reproduce manner with descriptions of the algorithms used. The structure of this chapter reflects the modular nature of the system itself by discussing the components individually.

The experimental process is further discussed in chapter 4. The results of the various experiments are presented.
In chapter 5, the experimental results are discussed and analysed. Conclusions and suggestions for future work are presented in Chapter 6.
Chapter 2

Review of Related Literature

Discussed in this chapter is a review of research works related to the specific requirements of the system presented in this project report. The chapter is divided into two main sections. The first section (2.1, Biological Depth Perception) introduces the fundamental biological framework of the human stereoscopic vision system. The second section (2.2, Computer Stereo Vision) is a review of related literature specific to hardware and software required in depth perception systems implemented in computers.

2.1 Biological Depth Perception

One major functionality of the human brain is the ability it has to construct a representation of the physical surroundings. To determine spatial relationships in three-dimensions, the brain must acquire a perception of the environment that is itself three dimensional. Individually, each eye only has the ability to perceive two-dimensions. However, when combined, the third dimension can be recovered [5]. Any animal with two eyes has stereoscopic vision, a term synonymous with “solid sight” referring to this three-dimensional perception of the world [1]. The following subsections discuss the processes involved in recovering this three-dimensional information.

2.1.1 Voluntary Eye Movement

There are three types of voluntary eye movement which are used to locate the point of gaze relative to the position of the head. These are pan, tilt and vergence [6].

The human eyes have the ability to look upward and downward relative
to the head position, in unison. This is the tilt mechanism. When eye tilt is combined with head and/or torso tilting, the Earth-vertical angle of rotation is limited only by the flexibility of an individual, however in the case of this research, the tilt angle is restricted to the servo-rotation angle of $90^\circ$.

Pan and vergence are both horizontal eye movements. The difference is that when the eyes are panning, they are rotating in the same direction, whereas, vergence implies that each eye is moving in the opposite direction to the other\(^1\). This can be either inward (convergence) or outward (divergence).

### 2.1.2 Saccades

Saccadic eye movements are small, involuntary, reflexive movements. These movements are supplementary to the voluntary pan, tilt and vergence movements. Saccades are very fast, with a peak angular speed of up to $900^\circ$ per second [7]. Their role is to gather additional information about the three-dimensional scene\(^2\).

### 2.1.3 Lens Accommodation

Accommodation is determined by the point-of-gaze. When the point-of-gaze is focused upon an object of interest (OoI) that is far away, the muscles around the eye’s lens are relaxed. The result is a thinner lens than if the OoI were closer, where the muscles would contract [5].

### 2.1.4 Stereopsis

The three physiological traits mentioned above are by themselves, just muscular movements. In order to build three-dimensional scene information, the brain combines information gathered by the eyes under effect of these movements. This is a mechanism known as stereopsis. The information is relative to two binocular cues (i.e., depth perception cues that require two eyes). The two cues are the binocular disparity and the vergence angle [1].

---

\(^1\)An exception to the statement that vergence implies opposite direction eye movement is when pan movement is occurring simultaneously, such that both eyes are rotating in the same direction, but at differing velocities.

\(^2\)Hardware limitations such as servo speed and the rate at which images can be processed make saccades non-viable in the proposed system.
Binocular Disparity

The binocular disparity is the difference in the images seen by each eye from their respective vantage points [1]. In general terms, the disparity between left and right corresponding points is greater when the object to which the point belongs is closer to the eyes than when at a greater distance. This assumes that the point-of-gaze is greater than or equal to distance than the object, or, that the eyes are not fixated on a particular point-of-gaze. The latter is only possible when the cyclopean angle of both eyes are equal – resulting in parallel line-of-sight vectors, which is not the normal function of stereo vision. If the object is at a greater distance than the point-of-gaze, the inverse will be true. I.e., The greater the difference in distance between the fixation point and the OoI point, the greater the disparity. Figure 2.1.4 shows the angles and distances used to define the binocular disparity of a point, $P$, with respect to a fixation point, $F$, at distance $d$ from the eye. $\Delta d$ is the distance between $F$ and $P$, $a$ is the baseline/interpupillary distance. Angle $\theta$ is the binocular subtense (the convergence angle) of $P$ and $\omega$ is the binocular subtense of $F$. [1].

Vergence

Vergence is a voluntary eye movement as mentioned above, but it also provides a binocular cue contributing to stereopsis.

Convergence and divergence of the eyes is a physiological trait that is useful in depth perception. It is the degree to which the two eyes converge upon a single point [5]. The point of fixation is considered to be the OoI as this is the point which will be at the centre of the field-of-view (FoV) of each eye. It will also be in the focus of the eye’s lens.

Vergence is known to enhance the perception of depth [8][9]. This means that while it is still possible for humans to estimate the depth of all objects in a scene, this perception will be most accurate at the point-of-gaze, i.e., the OoI.

2.1.5 Non-Stereo Depth Perception

Other depth perception cues exist where stereo vision is not a requirement. While not within the scope of the presented system, a brief discussion of these cues follows, to complete the overview of biological depth perception.
Figure 2.1: Binocular Disparity [1]
Motion Parallax

Motion parallax is the illusion of stereo vision caused by rapid, repetitive head movement. The result is a series of views attained in quick succession that can be used to gather similar information to that gathered by the binocular stereopsis cues. [10]

Shading Cues

Human knowledge of optics provides an additional depth perception cue. Using shading and specular highlights, the geometry and positioning of an object can be determined. This is perhaps the most important cue to humans. Particularly to those with only a single functional eye. Consider how a human views the world through a single eye; stereopsis is eliminated, however, depth perception is not lost entirely.

2.1.6 Field-of-View

The field-of-view (FoV), sometimes called the angle-of-view, is a property of each eye. It is the angular range of the visible scene at any one time. The human eye has an approximate FoV of 160° along the horizontal axis, and 120° on the vertical axis [11]. As established in section 3.1.2 of chapter 3, the horizontal FoV of the cameras used in this work is 75°.

2.1.7 Interpupillary Distance

The distance between the pupils of each eye is called the interpupillary or interocular distance. The average human interpupillary distance (IPD) is $65.46\text{mm} \pm 2.96\text{mm}$ [12]. This is useful in order to develop a biomimetic robotic system. However, a biomimetic IPD for the proposed system may have a negative effect on the accuracy of depth calculations when compared with larger IPDs. This is due to the inherent increase in similarity of the two views and the smaller, more precise vergence angles that will be required to track an object as it moves along the depth dimension ($z$).

2.2 Computer Stereo Vision

The hardware used for this research must mimic a biological vision system in that it should have the ability to track an object moving in three-dimensional space. The object of interest (OoI) must be the point of
gaze for both views at all times whilst the OoI is within the area of operation.

Stereo vision software is an important aspect of computer vision systems, with applications in aerial surveying and robotics, specifically in robot navigation and obstacle avoidance. But also in other larger-scale and more precision-critical systems such as NASA’s STEREO (Solar TErrestrial RElations Observatory) solar observation mission. The general idea is to replicate biological stereopsis in order to gather three-dimensional information of a scene.

Based on the information determined in section 2.1, it is known that the system must have the ability to control the angle of each camera for determining the depth of the OoI, an ability to pan left and right in order to track the object’s horizontal position and the ability to tilt up and down in order to track the object’s vertical position. Combined, there is a requirement for four degrees of freedom (DoF)\textsuperscript{3}.

### 2.2.1 Epipolar Geometry

Knowledge of epipolar geometry is fundamental to understanding stereo cameras. It describes the geometric relationship between two perspective cameras, either real or virtual – as discussed above. Both perspective cameras have each of the following epipolar components [13]:

**The Optical Centre**: is the is the centre of projection on the image plane ($O_L$ and $O_R$ in figure 2.2).

**The Epipolar Point**: is the point of intersection of the line joining the optical centres of both cameras (the baseline) with the image plane. I.e., The epipolar point is the image of the optical centre of the other camera ($E_L$ and $E_R$ in figure 2.2). [13].

**The Epipolar Plane**: is the plane defined by a given three-dimensional point ($P$ in figure 2.2) and the optical centre. The epipolar plane intersects the image plane, forming the image point ($P_L$ and $P_R$ in figure 2.2)[13].

\textsuperscript{3}The human eye also has the ability to tilt up and down, however, each eye is unable to achieve this independently. For this reason, the single tilt capability (representing the tilt of the head) is sufficient.
The Epipolar Line: is the straight line of intersection of the epipolar plane with the image plane. It is the image in one camera of a ray through the optical centre and image point in the other camera. All epipolar lines intersect at the epipolar point [13].

2.2.2 Real Camera Systems

In Kyriakoulis and Gasteratos [2], Marichal et al [14], Piater et al [15] and Samarawickrama and Sabatini [16] a robotic head with four servo motors for each of the discussed degrees of freedom is used. Each of these works has assembled their hardware with differing levels of robustness and with varied cost of fabrication whilst conforming to the same basic concept.

Figure 2.3 demonstrates a series of vergence angles for different OoI depths, as per the system presented in [2].

2.2.3 Virtual Camera Systems

In Yann et al [3], a different strategy for achieving gaze control using two views is presented. Instead of using two cameras, the left and right views are captured by a single camera pointed toward a central prism with two planar reflective surfaces. Two lateral plane mirrors are positioned either side of the prism. The stereo pair of views are reflected from the mirrors, onto the prism and captured from the perspective of the camera. Figure 2.4
Figure 2.3: Vergence Angles [2]

is a schematic top-view of this system.

The prism can be rotated around the point closest to the camera. This gives the system an eye gaze pan capability as illustrated by figure 2.5. $\gamma$ is the angle of rotation of the prism, the effect is a view pan rotation of $2\gamma$, but in the opposite direction. Although it was not implemented in [3], vergence control could be achieved by the rotation of the two lateral plane mirrors.

2.2.4 Vergence Control by Fuzzy Logic

In [2], a vergence controller that uses a fuzzy logic system with two inputs is presented. The input values are the Zero-Mean Normalised Cross Correlation (ZNCC) similarity measure and the depth estimation at each time step. The author claims the system to be computationally inexpensive and free from the requirement of a priori camera calibration.

The ZNCC similarity input is determined using equation 2.1.

$$ZNCC = 1 - \frac{\Sigma(I_r(u,v) - \bar{I}_r)(I_l(u,v) - \bar{I}_l)}{\sqrt{\Sigma(I_r(u,v) - \bar{I}_r)^2\Sigma(I_l(u,v) - \bar{I}_l)^2}} (2.1)$$

Where $I_r$ and $I_l$ are the right and left images, respectively. $\bar{I}_r$ and $\bar{I}_l$ are the mean values of the right and left images, respectively. $u$ and $v$ are the flattened UV map representation of the three-dimensional geometry of the scene.
Figure 2.4: Stereo Pair from a Reflective Prism [3]

Figure 2.5: Rotated Prism Creating a Pan Effect [3]
The depth estimation is an input requirement, which suggests that the system does not necessarily aim to acquire a precise depth measurement of an object of interest. This is a key difference between [2] and the vergence system proposed in chapter 3, such that the depth estimation is the resultant output of the vergence system. The output in [2] is the vergence angle itself. The depth estimation formula is shown by equation 2.2.

\[ Z = \frac{b}{2 \tan \theta} \]  

(2.2)

and

\[ \theta_{\text{vergence}} = 2 \theta \]  

(2.3)

Where \( Z \) is the depth of the scene, \( b \) is the baseline of the stereo head. \( \theta_{\text{vergence}} \) is the combined vergence angle, while \( \theta \) is the angle of one of the cameras. Therefore, this formula assumes symmetrical vergence only.

The ZNCC and \( Z \) values are fed into the fuzzy system in the form of gaussian membership functions, as per figures 2.6 and 2.7. The output is determined by the trapezoidal memberships function in figure 2.8.

![ZNCC Input Membership Function](image)

Figure 2.6: ZNCC Input Membership Function

“Sum” was used as the fuzzy aggregation operator for combining the generated truth for each rule in the rule base. The defuzzification method is the “Centroid” operator (i.e., the centre of the area under a curve). The
Figure 2.7: Depth Input Membership Function

rule base is shown in table 2.1

<table>
<thead>
<tr>
<th>Depth</th>
<th>Very Low</th>
<th>Low</th>
<th>Average</th>
<th>High</th>
<th>Very High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Near</td>
<td>DM</td>
<td>DM</td>
<td>D</td>
<td>D</td>
<td>Z</td>
</tr>
<tr>
<td>Near</td>
<td>DM</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>Z</td>
</tr>
<tr>
<td>Average</td>
<td>DM</td>
<td>D</td>
<td>D</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>Far</td>
<td>VM</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>Z</td>
</tr>
<tr>
<td>Very Far</td>
<td>VM</td>
<td>VM</td>
<td>V</td>
<td>V</td>
<td>Z</td>
</tr>
</tbody>
</table>

*DM=Diverge Much, D=Diverge, Z=Zero, V=Verge, VM=Verge Much*

In summary, the system presented in [2] provides an efficient vergence controller using fuzzy logic techniques. While the depth value generated at each step is not specific to a particular object, it is calculated according to the point-of-gaze of the cameras. The ZNCC similarity measure works in place of an object detection framework. The vergence angles are determined based on this similarity value where the output vergence angle is less as the similarity improves (the “Very High” column of table 2.1 has a value of “Zero” vergence for all depths). The result is that dominant objects in the scene contribute to the similarity. Therefore, if the point-of-gaze is toward the one of these objects, the similarity will be high. The author stated one possible area for
improvement due to ZNCCs susceptibility to noisy scenes where it may be impossible to get a high level of similarity, irrespective of the vergence angle.

2.3 Object Detection and Tracking

Research in object detection has been limited to a single framework. After a brief investigation of the possible methods, the Viola-Jones method described below became the obvious choice for the proposed system. The Viola-Jones method works in real time after training has been complete, it also produces rectangular areas surrounding detected objects, making it easy to work out the centre of the object. Finally, the frontal-face classifier which has been used is a pre-trained complex cascade structure and is suitable for the experimentation process, but may also be replaced with other classifiers with ease.

2.3.1 Viola-Jones Object Detection Framework

In [17], Viola and Jones propose a robust and fast face detection method using simple haar-like features to form frontal-face detection classifiers. The authors claim that their system is capable of face detection at approximately 15 times the speed of any previous approach, achieving 15 frames-per-second when published, using a 700MHz CPU.
Much like Haar basis features, the features used by the Viola-Jones framework are rectangular in shape, as shown in figure 2.9. The value of a feature at a position in the image is determined by the sum of the pixel values within the clear rectangles, subtracted from the sum of the pixels within the shaded rectangles. To achieve this summation in constant time, integral images (summed-area-tables) are used, whereby, any feature can be completely defined by looking up the values of the integral image at the rectangle corner locations within the feature. For example, features A and B in figure 2.9 each have two rectangles and the values can be computed in six look-ups. Feature C has three rectangles and can be computed in eight look-ups and D has four rectangles and can be computed in nine look-ups. The integral image at pixel location \((x, y)\) contains the sum of the pixels above and to the left of \((x, y)\), inclusive.

![Figure 2.9: Rectangular Features used by the Viola-Jones Object Detection Framework](image)

The object detection framework uses the learning algorithm AdaBoost to select the best features and to train the strong classifiers. Algorithm 1 demonstrates the boosting algorithm for learning a query online. \(T\) hypotheses are constructed each using a single feature. The final hypothesis is a weighted linear combination of the \(T\) hypotheses where the weights are inversely proportional to the training errors [17].
Algorithm 1 AdaBoost Algorithm for the Viola-Jones Object Detection Framework

Require: example images \((x_1, y_1), ..., (x_n, y_n)\) where \(y_i = 0, 1\) for negative and positive examples respectively.
Initialise weights \(w_{1,i} = \frac{1}{2m}, \frac{1}{2l}\) for \(y_i = 0, 1\) respectively, where \(m\) and \(l\) are the number of negatives and positives respectively.

for all \(t = 1, ..., T\) do

Normalise the weights, \(w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}\)

Select the best weak classifier with respect to the weighted error, \(\epsilon_t = \min_{f,p,\theta} \sum_i w_i |h(x_i, f, p, \theta) - y_i|\)

Define \(h_t(x) = h(x, f_t, p_t, \theta_t)\) where \(f_t, p_t\) and \(\theta_t\) are the minimisers of \(\epsilon_t\).

Update the weights, \(w_{t+1,i} = w_{t,i} \beta_t^{1-e_i}\) where \(e_i = 0\) if example \(x_i\) is classified correctly, \(e_i = 1\) otherwise, and \(\beta_t = \frac{\epsilon_t}{1-\epsilon_t}\).

end for

The final strong classifier is:

\[ C(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases} \]

where \(\alpha_t = \log \frac{1}{\beta_t}\).
In order to evaluate the strong classifiers generated by the AdaBoost process, they are arranged in a cascade, in order of complexity. For example, the simpler classifiers are evaluated first for each sub-window, rejecting the majority without the need for further processing. A sub-window that cascades through all of the classifiers is identified as a face. If at any stage throughout the cascading process, any classifier rejects the sub-window, then no further processing occurs and the next sub-window is passed through the cascade.

The cascading classifiers were improved by Lienhart, et al in [18]. These cascade classifier tree XML files are included in the OpenCV library for easy implementation.
Chapter 3

Proposed System

The system proposed by this research involves both the fabrication of a simple proof-of-concept hardware set-up and a driving software platform. Discussed in this chapter are the sub-components of these two main factors.

3.1 Hardware

High precision stereo camera heads are available for purchase, however, the market price for equipment of this nature is somewhat extravagant considering the purpose of this research. Given that this work aims to establish a comparative analysis of depth perception methods rather than the hardware itself, a simple set-up employing two USB web cameras and four hobby servos is sufficient.

3.1.1 Design of Hardware

The system is to be biomimetic with respect to the human visual system, such that the movement of cameras is to emulate that of eyes. Moreover, the system should allow for pan and tilt motion, representing the ability humans have to rotate the viewing area through neck movement. As discussed in section 2.2, humans are able to tilt the area of view with both the neck and eyes. These two mechanisms are represented by a single tilting servo in this work. Figure 3.1 illustrates the concept design of the servo/camera set-up.

3.1.2 Fabrication

Having established a design for the hardware, the next stage is to source suitable cameras and servos as well as a servo controller for the purpose
of the proposed system. Figure 3.2 shows the assembled hardware. The individual components are described in the following paragraphs.

Camera

The Logitech\textsuperscript{TM} QuickCam\textsuperscript{TM} Pro 9000 (see figure 3.3) has been selected as the camera device used in this work. Any USB web camera would have been sufficient, however, this model was readily available.

The camera has a horizontal field-of-view (FoV) of \( \approx 75^\circ \), determined empirically by situating the camera at a fixed distance from a wall \((z)\). The length of the wall visible in the camera image is divided by \(2\) \((x)\) (See figure 3.4). Given \(x\) and \(z\) the FoV can be calculated as per equation 3.1.

\[
FoV = 2\alpha = 2\tan^{-1}(x/z)
\]  

Servos

Four servos are required in order to enable four degrees-of-freedom (4DoF). Two are utilised for vergence control (I.e., independent pan motion for each camera), one for the head-tilt motion and the other for head-pan motion. Two Hextronik HXT900 servos have been selected for the vergence. These popular micro servos are suitable for the load of a single web camera, however, when this same model servo was used for pan and tilt control, it was found that they could not draw enough current to adequately move the
Figure 3.2: Assembled Servo and Camera Hardware

Figure 3.3: Logitech™ QuickCam™ Pro 9000
aluminium based set-up. For this reason, two larger servos are used for pan and tilt.

The specifications of the HXT900 that are required by the software in order to determine position and angle values are shown in table 3.1.

**Servo Controller**

The servo controller selected for this project is the Pololu Micro Maestro 6-Servo USB Micro-Controller. As the name suggests, it is connected via USB to the host machine where the software is executed. It has the ability to control up to six servos. Seeing as only four servos are required this controller is adequate for the purpose. Figure 3.6 is a photograph of the controller.
Figure 3.5: Hextronik HXT900 9 gram Servo

Figure 3.6: Pololu Micro Maestro 6-Servo Micro-Controller
3.2 Software

3.2.1 Serial Servo Interface

The serial interface for the Pololu device appears as two virtual ports when connected via USB. The command port (usually /dev/ttyACM0) and the TTL port (usually /dev/ttyACM1). There are three serial mode configurations; USB Dual Port, USB Chained and UART.

The USB Dual port mode is used in the proposed system. In this mode, the command port can be used to send commands to the Maestro and receive responses from it. The baud rate set in the program when opening the command port is irrelevant. The TTL port can be used to send bytes on the TX line and receive bytes on the RX line. The baud rate set in the program when opening the TTL port determines the baud rate used to receive and send bytes on RX and TX. This allows the computer to control the Maestro and simultaneously use the RX and TX lines as a general purpose serial port that can communicate with other types of TTL serial devices [4]. Figure 3.7 is a diagram of the host to device communication using this mode.

The device is compatible with three protocols. The Compact Protocol, the Pololu Protocol and the mini SSC Protocol. The compact protocol is the recommended protocol by pololu [4]. This protocol has the basic packet form of a command byte followed by any necessary data bytes. The required functionality of the serial interface is as follows:

- **Set Target**: This command is used to set the position of of a servo. The compact protocol for “set target” is 0x84, *channel number, target low bits, target high bits*. 0x84 is the command byte, the channel number is a byte indicating the number of the servo to be moved. The lower 7-bits of each of the final two bytes contain the target position. The third byte contains bits 0-6 and the fourth byte contains bits 7-14. The most significant bit of these two bytes is irrel-
evant. The position is in quarter-microsecond units, representing the pulse width. If the position is set to 0, this signals the controller to stop sending pulses to the servo, i.e., turn the servo off.

- **Set Speed:** This command is used to set the maximum speed of a servo. The compact protocol for “set speed” is 0x87, channel number, speed low bits, speed high bits. Similar to set target, the third and fourth bytes use only the lower 7-bits. The speed limit is in units of \((0.25 \mu s)/(10 \text{ ms})\). If the speed is set to 0, this is treated as unlimited speed, the proposed system never limits the servo speed.

- **Set Acceleration:** This command is used to set the acceleration of a servo. The compact protocol for “set acceleration” is 0x89, channel number, acceleration low bits, acceleration high bits. The acceleration limit is in units of \((0.25 \mu s)/(10 \text{ ms})/(80 \text{ ms})\). As with set speed, a value of 0 represents no limit. The minimum value is 1, and the maximum is 255. However, the proposed system does not limit the acceleration, a value of 0 is always set.

- **Get Position:** The proposed system is capable of getting the position in two ways. Either from a value in memory that is set each time setTarget(...) is called or by the protocol command 0x90, channel number. The result of this command is that two bytes are written to the device file, the position low bits and the position high bits. These will correspond to the most recent values as given by set target. As such, the first method for getting the servo position is always used, for efficiency.

- **Get Speed:** There is no protocol command for getting the speed, this functionality simply returns the speed from the value stored in memory. It is only included for completeness.

- **Get Acceleration:** As with get speed, get acceleration is only included for completeness.

- **Reset Servos:** A function to restore all of the default servo positions has been included, it simply calls the setTarget(...) for each connected servo, passing a target as defined for a DEFAULT_SERVO_POSITION macro. This value is set to the middle of the position pulse width range, 1496 microseconds.

- **Shutdown Servos:** At the termination of the program, the servos will continue to be sent the most recent target pulse, as such, the destructor
calls a `shutdownServos()` method which sends a target of 0 to each connected servo.

Communication between the software and hardware controller is handled with the `unistd` header of the C POSIX library and the `termios` Unix API for terminal input/output. This restricts the program to Unix-based environments.

The serial servo interface is implemented as a C++ class (`SerialServoController` – See Appendix A. on page 54). All of the aforementioned functionalities have corresponding methods within the class. To convert a value to the two bytes required by the `setTarget(...)` method, the target value (an integer) is passed to two additional functions, both of which return an unsigned char, i.e., a single byte:

- `unsigned char SerialServoController::getLower7Bits(int value)`
  performs a bitwise logical conjunction of the value and the hexadecimal constant `0x7F` (i.e., `01111111` in binary). The result is cast to an unsigned char and returned.

- `unsigned char SerialServoController::getHigher7Bits(int value)`
  performs a 7-bitwise right shift of the value then performs the logical conjunction with `0x7F`.

The compact protocol commands are sent to the device through the `ssize_t write(int fd, const void *buf, size_t count)` function in the `unistd.h` header. Where `fd` is the file descriptor of the open device file. *buf is a pointer to the first byte of the command and count is the number of bytes in the command.

### 3.2.2 Camera Interface

The OpenCV computer vision library has served as the underlying interface to the two Logitech cameras. A C++ class (`StereoCamera` – See Appendix B. on page 60), captures frames from both cameras using the OpenCV functions inside a `getFrame()` method. This class is designed to be inherited by the object recognition classes discussed in the next subsection.
3.2.3 Object Identification and Tracking

For the purpose of the experiments undertaken in this work, the Viola-Jones Object Detection Framework has been implemented. More specifically, the frontal-face detection variant. The software is modular and has been designed in such a way that it is simple to substitute the object detection class for any other algorithm.

The FaceDetection class (See Appendix C. on page 63) inherits from StereoCamera so it can access and process captured frames. It contains a modified version of the getFrame() method that performs the face detection algorithm each time it is called. Following this, the centre point of the best detected face in each frame image is stored in memory. These are the values used to determine the next point of gaze by the vergence controller.

The Viola-Jones face detector as discussed in section 2.3.1 on page 18 is implemented for both of the cameras. The OpenCV function cvHaarDetectObjects() implements this cascade detection algorithm, and has been used in this work. The pre-trained frontal-face classifier distributed with OpenCV (Developed in [18]) is used in the vergence and depth experimentation process in the next chapter.

3.2.4 Vergence Angle Controller

Two methods for determining the required vergence angle of the two cameras are proposed. Each of these two methods has an expected advantage over the other. The experimental process in chapter 4 attempts to determine the degree of these advantages.

Gradual Vergence

The first, and simplest vergence control method is one where a constant stream of packets are sent to the servo controller. Each packet moves the servos closer to the desired point of gaze. The expectation is, for a stationary target, the accuracy of this method will be very high. The disadvantage however, is that the vergence controller will constantly chase a moving target without reaching an exact convergence point.

The target positions of the two vergence servos are determined by equa-
tion 3.2.

\[ \text{pos}_{\text{new}} = \begin{cases} \text{pos}_{\text{old}} - (\text{OoI}_x - \text{halfWidth}) & \text{if } \text{OoI}_x > \text{halfWidth} \\ \text{pos}_{\text{old}} + (\text{halfWidth} - \text{OoI}_x) & \text{if } \text{OoI}_x < \text{halfWidth} \\ \text{pos}_{\text{old}} & \text{otherwise} \end{cases} \] (3.2)

Where \( \text{OoI}_x \) is the column location of the object of interest in pixels, \( \text{halfWidth} \) is the width of the image divided by two, in pixels and \( \text{pos}_{\text{old}} \) is the current position of the servo in \( \mu s/4 \). All this does is move the servos very gradually, until the desired point of convergence is met. While it can be used on its own, the primary purpose of this controller is simply to provide an accurate initial measurement for the geometric vergence controller discussed next.

**Geometric Vergence**

In the second vergence controller (and main focus of this project), convergence is achieved from inverse trigonometric calculations. This requires knowledge of the field of view (FoV) as discussed above in section 3.1.2. The basic principle is that given the location of the OoI in each camera, the controller can determine the offset position for each vergence servo. This is achieved using the known geometric properties for each camera. For instance, the FoV angle is known, therefore, at any given depth, the horizontal dimension in world units can be determined.

Algorithm 2 is used to determine the vergence angles of each camera servo. Note the requirement for previousDepth. In order to obtain the initial depth value, the gradual vergence controller can be used to find the OoI for the first time, after this depth is determined, gradual vergence is not needed again. Alternatively, the depth and initial vergence angles could be explicitly instantiated to any correct value.

The previous depth measurement (\( \text{previousDepth} \)), the pixel width of a single-camera frame (\( \text{frameWidth} \)), the horizontal centre of the object of interest in both camera frames (\( \text{leftCentroid.x} \) and \( \text{rightCentroid.x} \)), the current vergence servo position in quarter-microseconds (\( \text{leftServoPosition} \) and \( \text{rightServoPosition} \)) and the field of view angle for the single-camera lens (\( \text{FoV} \)) are required by the vergence controller. The first step is to calculate the length of half of the visible horizontal image at the previous known depth in world units (\( \text{opposite} \)). From the frame width, the horizontal centre of the frame can be determined (\( \text{halfWidth} \)). Using this value and the current pixel location of the OoI, the horizontal offset in pixels of the OoI in both cameras
can be determined \((\text{leftOffset} \text{ and } \text{rightOffset})\). These offsets can then be scaled to world units by using the measured \textit{opposite} value (i.e., converted to the same unit the baseline is recorded in). These scaled offsets determine the required rotation of vergence angles trigonometrically \((\text{leftRotateDegrees} \text{ and } \text{rightRotateDegrees})\). The angles then need to be converted to servo pulse-width positions \((\text{leftPositionOffset} \text{ and } \text{rightPositionOffset})\). Finally, the new servo positions can be set by adding or subtracting the offsets to the current positions (depending on if divergence or convergence is required). Given the new point-of-gaze, the depth can now be calculated.

The \texttt{int degreesToServoPosition(float angle)} method returns a servo target position by multiplying angle by a predetermined constant, \texttt{QUARTER_MICROSECONDS_PER_DEGREE}. This constant is determined by equation 3.3

\[
\frac{1}{4} \mu s/\text{deg} = \frac{\text{servo}_{\text{max}} - \text{servo}_{\text{min}}}{\text{servo}_{\text{range}}} \tag{3.3}
\]

Where \text{servo}_{\text{max}} and \text{servo}_{\text{min}} in the case of the Hextronik servos are \(2000 \times 4\) and \(992 \times 4\) respectively and \text{servo}_{\text{range}} is \(90.0^\circ\).

The \texttt{float calculateDepth()} method is illustrated by algorithm 4 in section 3.2.6.

### 3.2.5 Pan and Tilt Control

The servo panning algorithm aims to rotate the robotic head to a position where the two vergence angles are near-symmetrical. This is similar to how a human would look around a scene. For instance, a person does not often look around by voluntary eye movements alone. The pan mechanism represents neck rotation where the whole head is oriented toward the OoI. When the pan servo has reached the maximum or minimum position, it will not be able to guarantee symmetrical vergence angles, much like a human head when the neck is at its maximum rotation, left or right and the point-of-gaze is not in the exact direction of the head orientation.

The tilt controller is the only component that requires the vertical coordinate of the OoI. For simplicity, this coordinate is taken from the left camera only. Being that the two cameras will always tilt in unison and assuming the system is situated on a flat surface, the \(OoI.y\) value in each camera frame will be approximately the same at all times.
Algorithm 2 Geometric Vergence Controller

Require: previousDepth, frameWidth, leftCentroid.x, rightCentroid.x, leftServoPosition, rightServoPosition, FoV

\[
\text{opposite} \leftarrow \text{previousDepth} \times \tan\left(\frac{\text{FoV}}{2}\right)
\]
\[
\text{halfWidth} \leftarrow \text{frameWidth} / 2
\]

if leftCentroid.x < halfWidth then
    leftOffset \leftarrow \text{halfWidth} - \text{leftCentroid.x}
else
    leftOffset \leftarrow \text{halfWidth} - (\text{frameWidth} - \text{leftCentroid.x})
end if

if rightCentroid.x < halfWidth then
    rightOffset \leftarrow \text{halfWidth} - \text{rightCentroid.x}
else
    rightOffset \leftarrow \text{halfWidth} - (\text{frameWidth} - \text{rightCentroid.x})
end if

leftOffset \leftarrow \text{opposite} \times \left(\frac{\text{leftOffset}}{\text{halfWidth}}\right)
rightOffset \leftarrow \text{opposite} \times \left(\frac{\text{rightOffset}}{\text{halfWidth}}\right)

leftRotateDegrees \leftarrow \tan^{-1}\left(\frac{\text{leftOffset}}{\text{previousDepth}}\right) \times \left(\frac{180}{\pi}\right)
rightRotateDegrees \leftarrow \tan^{-1}\left(\frac{\text{rightOffset}}{\text{previousDepth}}\right) \times \left(\frac{180}{\pi}\right)

leftPositionOffset \leftarrow \text{degreesToServoPosition}(\text{leftRotateAngle})
rightPositionOffset \leftarrow \text{degreesToServoPosition}(\text{rightRotateAngle})

if leftCentroid.x > halfWidth then
    leftServoPosition \leftarrow leftServoPosition - leftPositionOffset
else if leftCentroid.x < halfWidth then
    leftServoPosition \leftarrow leftServoPosition + leftPositionOffset
end if

if rightCentroid.x > halfWidth then
    rightServoPosition \leftarrow rightServoPosition - rightPositionOffset
else if rightCentroid.x < halfWidth then
    rightServoPosition \leftarrow rightServoPosition + rightPositionOffset
end if

previousDepth \leftarrow \text{calculateDepth()}
Algorithm 3 demonstrates the servo pan and tilt control method. Admittedly, this is quite a naive implementation, similar to the gradual vergence controller. However, it has no adverse effect on the accuracy of depth perception.

**Algorithm 3 Pan and Tilt Control**

**Require:** frameHeight, OoI.x, OoI.y, tiltServoPosition, panServoPosition

degLeft ← 90.0 - getCyclopeanAngle(LEFT_EYE)
degRight ← 90.0 - getCyclopeanAngle(RIGHT_EYE)

if OoI.y > (frameHeight / 2) then
    tiltServoPosition ← tiltServoPosition - (OoI.y - (frameHeight / 2))
else if OoI.y < (frameHeight / 2) then
    tiltServoPosition ← tiltServoPosition + (frameHeight / 2) - OoI.y
end if

if degLeft > degRight then
    panServoPosition ← panServoPosition + (OoI.x - (frameWidth / 2))
else if degLeft < degRight then
    panServoPosition ← panServoPosition - (frameWidth / 2) - OoI.x;
end if

3.2.6 Depth Perception and Calculation

The simple technique in algorithm 4 is used to triangulate the depth of the OoI from the point-of-gaze (vergence angles) and the baseline IPD.

**Algorithm 4 Calculate Depth**

**Require:** baseline

degLeft ← 90.0 - getCyclopeanAngle(LEFT_EYE)
degRight ← 90.0 - getCyclopeanAngle(RIGHT_EYE)
radiansLeft ← degLeft × (π/180.0)
radiansRight ← degRight × (π/180.0)
tanThetaLeft ← tan(radiansLeft)
tanThetaRight ← tan(radiansRight)
return baseline × ((tanThetaLeft × tanThetaRight) / (tanThetaLeft + tanThetaRight))

The float getCyclopeanAngle(int eye) method determines the angle of a servo based on its position relative to the neutral angle (directly ahead).
Equation 3.4 demonstrates this process.

\[ cyclopean\, Angle = \frac{|servo_{position} - servo_{neutral}|}{\frac{1}{4} \mu s/deg} \]  

(3.4)

Where \( servo_{position} \) is the current position of a servo, \( servo_{neutral} \) is the zero degrees angle of the servo and \( \frac{1}{4} \mu s/deg \) is the value as determined by equation 3.3, above.

3.2.7 Class Structure

Figure 3.8 shows the class structure of the proposed system software. See the Appendices for their complete implementation.

3.3 Summary

The proposed system is a built-for-purpose hardware set-up and software controller that has the functionality of a human head, such that it can follow a moving face around a scene (within the limits of the servos). The geometric vergence controller is a precise method of rotating the point of gaze toward an object of interest. The advantage of this method is a low serial controller throughput compared with other vergence controllers that use fuzzy logic and imprecise values. In the next chapter, the results of experiments conducted with this system are presented.
Figure 3.8: The UML Class Diagram of the Proposed System
Chapter 4

Results

The two different convergence methods have been tested independently for two baseline configurations. The raw data is presented in tabular form and as a scatter chart indicating the accuracy of the vergence angles at different distances. An additional chart, plotting the mean and standard deviation is presented for each experiment, this illustrates the reliability of a depth measurement at known ‘actual’ distances.

The depth measurements have been taken for eighteen distances between a minimum of 15 cm and a maximum of 150 cm. Measurements are taken for stationary targets only and have not been taken in chronological order so as to avoid patterns which may be owing to small divergence as the OoI moves further away. This is particularly important to the geometric vergence method, where the parallax between the two cameras is critical to the calculation of the required vergence angles. It is expected that if the required divergence or convergence is only small, the accuracy is likely to be much higher. Therefore, prior to making the next distance measurement in a sample, the OoI is moved to a position dissimilar to the next measurement location.

4.1 Gradual Vergence

To calibrate the gradual convergence experiments, the two cameras must first be parallel to each other in their default positions. This is achieved by manual adjustments and verified by allowing the cameras to converge to a target object at a known distance, close to the camera (as preliminary results suggested short distances produced the highest accuracy). Once the cameras are correctly set up, the head is reset and experiments can
In the presented experiments, the head is re-adjusted after every experiment sample. A total of five samples at eighteen different distances are measured in each set of results. Two different baseline distances have been tested. First, 100 mm, the closest possible distance permitted by the hardware dimensions – this is the closest representation of the biological interpupillary distance (IPD) of 60mm – 70mm. 250 mm, used to investigate if results improve with a greater baseline. It was not possible to measure at distances less than the human average IPD due to the size of the cameras.

4.1.1 100 mm Baseline
For a baseline of 100 mm, using the gradual controller, the experimental results are shown in table 4.1 and figure 4.1. For short distances (< 45cm), the results appear to have a high accuracy and low error. However, for all measurements beyond this depth, there is a noticeable underestimation of the depth and a large increase in error. Figure 4.2 shows the sample mean with error bars for each distance. Notice the ratio between the x and y axes is not the expected (1:1). For example, the trend line at the real depth distance of 150cm is barely above 120cm on the estimated depth axis.

![Depth From Gradual Convergence](image)

Figure 4.1: Gradual Vergence (100 mm baseline) X Y Scatter Plot for Five Samples at Eighteen Distances
Table 4.1: Gradual Vergence (100 mm baseline) Experimental Results

<table>
<thead>
<tr>
<th>cm</th>
<th>Sample 1</th>
<th>Sample 2</th>
<th>Sample 3</th>
<th>Sample 4</th>
<th>Sample 5</th>
<th>Mean</th>
<th>Std. Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>15.044</td>
<td>14.996</td>
<td>15.724</td>
<td>16.128</td>
<td>14.960</td>
<td>15.4</td>
<td>0.5</td>
</tr>
<tr>
<td>20</td>
<td>19.766</td>
<td>19.654</td>
<td>19.782</td>
<td>19.995</td>
<td>20.128</td>
<td>19.9</td>
<td>0.2</td>
</tr>
<tr>
<td>25</td>
<td>25.476</td>
<td>26.070</td>
<td>24.496</td>
<td>26.880</td>
<td>25.325</td>
<td>25.6</td>
<td>0.9</td>
</tr>
<tr>
<td>30</td>
<td>31.290</td>
<td>29.804</td>
<td>28.765</td>
<td>29.772</td>
<td>27.480</td>
<td>29.4</td>
<td>1.4</td>
</tr>
<tr>
<td>35</td>
<td>32.781</td>
<td>33.469</td>
<td>34.400</td>
<td>33.980</td>
<td>35.016</td>
<td>33.9</td>
<td>0.9</td>
</tr>
<tr>
<td>40</td>
<td>36.080</td>
<td>38.641</td>
<td>37.995</td>
<td>32.764</td>
<td>38.881</td>
<td>36.9</td>
<td>2.5</td>
</tr>
<tr>
<td>45</td>
<td>43.258</td>
<td>40.440</td>
<td>41.259</td>
<td>42.016</td>
<td>43.867</td>
<td>42.2</td>
<td>1.4</td>
</tr>
<tr>
<td>50</td>
<td>44.265</td>
<td>40.789</td>
<td>40.987</td>
<td>41.659</td>
<td>43.560</td>
<td>42.3</td>
<td>1.6</td>
</tr>
<tr>
<td>60</td>
<td>47.884</td>
<td>57.889</td>
<td>60.789</td>
<td>51.486</td>
<td>56.500</td>
<td>54.9</td>
<td>5.2</td>
</tr>
<tr>
<td>70</td>
<td>61.121</td>
<td>64.784</td>
<td>65.489</td>
<td>62.178</td>
<td>66.366</td>
<td>64.0</td>
<td>2.2</td>
</tr>
<tr>
<td>80</td>
<td>64.128</td>
<td>66.788</td>
<td>67.948</td>
<td>69.814</td>
<td>77.164</td>
<td>69.2</td>
<td>4.9</td>
</tr>
<tr>
<td>90</td>
<td>50.125</td>
<td>69.994</td>
<td>75.224</td>
<td>79.365</td>
<td>81.256</td>
<td>71.2</td>
<td>12.5</td>
</tr>
<tr>
<td>100</td>
<td>84.400</td>
<td>89.658</td>
<td>79.885</td>
<td>78.985</td>
<td>84.988</td>
<td>83.6</td>
<td>4.3</td>
</tr>
<tr>
<td>110</td>
<td>100.250</td>
<td>89.564</td>
<td>99.557</td>
<td>88.561</td>
<td>120.065</td>
<td>99.6</td>
<td>12.7</td>
</tr>
<tr>
<td>120</td>
<td>100.540</td>
<td>110.000</td>
<td>87.565</td>
<td>88.569</td>
<td>79.651</td>
<td>93.3</td>
<td>12.0</td>
</tr>
<tr>
<td>130</td>
<td>131.915</td>
<td>116.870</td>
<td>91.550</td>
<td>99.154</td>
<td>94.568</td>
<td>106.8</td>
<td>17.1</td>
</tr>
<tr>
<td>140</td>
<td>130.564</td>
<td>141.125</td>
<td>101.548</td>
<td>97.595</td>
<td>102.560</td>
<td>114.7</td>
<td>19.8</td>
</tr>
<tr>
<td>150</td>
<td>108.255</td>
<td>135.654</td>
<td>121.054</td>
<td>140.400</td>
<td>100.967</td>
<td>121.3</td>
<td>17.0</td>
</tr>
</tbody>
</table>
Figure 4.2: Gradual Vergence (100 mm baseline) Mean Values and Error Bars
4.1.2 250 mm Baseline

Table 4.2 contains the raw data gathered from measurements using the gradual vergence method with a 250 mm IPD baseline. For each of the eighteen measured distances, the mean average and the standard deviation over five samples is also included. Figure 4.3 plots the results for each sample. The mean and error are shown in figure 4.4. This experiment produced a highly accurate measurement for all distances. The standard deviation for distance measurements did increase slightly with the actual distance. However, there is no severe underestimation as with the previous experiment using a 100 mm baseline.

<table>
<thead>
<tr>
<th>cm</th>
<th>Sample 1</th>
<th>Sample 2</th>
<th>Sample 3</th>
<th>Sample 4</th>
<th>Sample 5</th>
<th>Mean</th>
<th>Std. Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>15.666</td>
<td>16.126</td>
<td>16.058</td>
<td>16.647</td>
<td>15.131</td>
<td>15.9</td>
<td>0.6</td>
</tr>
<tr>
<td>20</td>
<td>20.964</td>
<td>21.060</td>
<td>21.072</td>
<td>21.660</td>
<td>19.964</td>
<td>20.9</td>
<td>0.6</td>
</tr>
<tr>
<td>30</td>
<td>30.929</td>
<td>30.439</td>
<td>30.039</td>
<td>31.801</td>
<td>30.001</td>
<td>30.6</td>
<td>0.7</td>
</tr>
<tr>
<td>35</td>
<td>34.126</td>
<td>35.722</td>
<td>35.103</td>
<td>35.290</td>
<td>36.782</td>
<td>35.4</td>
<td>1.0</td>
</tr>
<tr>
<td>40</td>
<td>39.752</td>
<td>40.331</td>
<td>40.192</td>
<td>41.520</td>
<td>40.444</td>
<td>40.4</td>
<td>0.7</td>
</tr>
<tr>
<td>45</td>
<td>42.550</td>
<td>46.349</td>
<td>45.625</td>
<td>45.672</td>
<td>45.002</td>
<td>45.0</td>
<td>1.5</td>
</tr>
<tr>
<td>50</td>
<td>49.637</td>
<td>50.808</td>
<td>51.728</td>
<td>50.360</td>
<td>50.100</td>
<td>50.5</td>
<td>0.8</td>
</tr>
<tr>
<td>60</td>
<td>57.488</td>
<td>57.708</td>
<td>61.757</td>
<td>60.484</td>
<td>61.241</td>
<td>59.7</td>
<td>2.0</td>
</tr>
<tr>
<td>70</td>
<td>66.911</td>
<td>67.787</td>
<td>70.514</td>
<td>70.326</td>
<td>71.323</td>
<td>69.4</td>
<td>1.9</td>
</tr>
<tr>
<td>80</td>
<td>70.548</td>
<td>78.358</td>
<td>80.219</td>
<td>77.088</td>
<td>80.949</td>
<td>77.4</td>
<td>4.1</td>
</tr>
<tr>
<td>90</td>
<td>91.527</td>
<td>86.029</td>
<td>89.099</td>
<td>88.262</td>
<td>90.262</td>
<td>89.0</td>
<td>2.1</td>
</tr>
<tr>
<td>100</td>
<td>99.992</td>
<td>98.977</td>
<td>101.510</td>
<td>95.898</td>
<td>102.620</td>
<td>99.8</td>
<td>2.6</td>
</tr>
<tr>
<td>110</td>
<td>107.385</td>
<td>111.652</td>
<td>110.673</td>
<td>103.149</td>
<td>112.013</td>
<td>109.0</td>
<td>3.7</td>
</tr>
<tr>
<td>120</td>
<td>124.916</td>
<td>110.000</td>
<td>118.101</td>
<td>120.861</td>
<td>121.490</td>
<td>119.1</td>
<td>5.6</td>
</tr>
<tr>
<td>130</td>
<td>131.915</td>
<td>116.870</td>
<td>130.804</td>
<td>129.900</td>
<td>132.420</td>
<td>128.4</td>
<td>6.5</td>
</tr>
<tr>
<td>140</td>
<td>141.904</td>
<td>138.825</td>
<td>135.572</td>
<td>124.613</td>
<td>140.060</td>
<td>136.2</td>
<td>6.9</td>
</tr>
<tr>
<td>150</td>
<td>152.446</td>
<td>160.595</td>
<td>150.824</td>
<td>147.826</td>
<td>148.690</td>
<td>152.1</td>
<td>5.1</td>
</tr>
</tbody>
</table>
Figure 4.3: Gradual Vergence (250 mm baseline) X Y Scatter Plot for Five Samples at Eighteen Distances

Figure 4.4: Gradual Vergence (250 mm baseline) Mean Values and Error Bars
4.2 Geometric Vergence

Unlike the gradual vergence method, the measurements taken for the geometric method are after a single data packet is sent to each servo. This of course makes it likely that the produced results will be less accurate. The point of these experiments however, is to measure just how much the accuracy is affected due to the reduction in the quantity of data packets required to achieve convergence. As mention above, the measurements are taken in non-chronological order. This causes the \( \Delta \) degree-of-vergence to be different for each measurement.

4.2.1 100 mm Baseline

Using the geometric controller, for a baseline of 100 mm, the results are more accurate than expected having already performed the gradual vergence experiment at a 100 mm baseline. The raw results are shown in table 4.3 and figure 4.5. The standard deviation does increase significantly as the actual distance is increased, but there is not the same problem of severe underestimation as with the gradual controller. Figure 4.6 displays the sample means and error bars for this experiment.

Figure 4.5: Geometric Vergence (100 mm baseline) X Y Scatter Plot for Five Samples at Eighteen Distances
<table>
<thead>
<tr>
<th>cm</th>
<th>Sample 1</th>
<th>Sample 2</th>
<th>Sample 3</th>
<th>Sample 4</th>
<th>Sample 5</th>
<th>Mean</th>
<th>Std. Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>15.987</td>
<td>16.112</td>
<td>15.655</td>
<td>15.987</td>
<td>14.999</td>
<td>15.7</td>
<td>0.5</td>
</tr>
<tr>
<td>20</td>
<td>21.025</td>
<td>20.265</td>
<td>19.265</td>
<td>19.754</td>
<td>18.786</td>
<td>19.8</td>
<td>0.9</td>
</tr>
<tr>
<td>30</td>
<td>30.158</td>
<td>30.569</td>
<td>31.025</td>
<td>31.569</td>
<td>29.893</td>
<td>30.6</td>
<td>0.7</td>
</tr>
<tr>
<td>35</td>
<td>33.669</td>
<td>34.469</td>
<td>37.569</td>
<td>35.895</td>
<td>33.878</td>
<td>35.1</td>
<td>1.6</td>
</tr>
<tr>
<td>40</td>
<td>35.489</td>
<td>38.465</td>
<td>42.564</td>
<td>41.651</td>
<td>36.560</td>
<td>38.9</td>
<td>3.1</td>
</tr>
<tr>
<td>45</td>
<td>42.650</td>
<td>45.100</td>
<td>44.697</td>
<td>41.567</td>
<td>48.564</td>
<td>44.5</td>
<td>2.7</td>
</tr>
<tr>
<td>50</td>
<td>47.420</td>
<td>44.450</td>
<td>51.260</td>
<td>60.210</td>
<td>49.883</td>
<td>50.6</td>
<td>5.9</td>
</tr>
<tr>
<td>60</td>
<td>55.206</td>
<td>51.526</td>
<td>66.260</td>
<td>57.604</td>
<td>52.190</td>
<td>56.6</td>
<td>5.9</td>
</tr>
<tr>
<td>70</td>
<td>70.265</td>
<td>74.250</td>
<td>62.044</td>
<td>66.548</td>
<td>59.787</td>
<td>66.6</td>
<td>5.9</td>
</tr>
<tr>
<td>80</td>
<td>69.888</td>
<td>78.232</td>
<td>74.500</td>
<td>89.540</td>
<td>81.307</td>
<td>78.7</td>
<td>7.4</td>
</tr>
<tr>
<td>90</td>
<td>95.561</td>
<td>81.345</td>
<td>88.567</td>
<td>79.150</td>
<td>76.598</td>
<td>84.2</td>
<td>7.7</td>
</tr>
<tr>
<td>100</td>
<td>85.644</td>
<td>81.650</td>
<td>87.540</td>
<td>115.650</td>
<td>102.396</td>
<td>94.6</td>
<td>14.2</td>
</tr>
<tr>
<td>110</td>
<td>95.654</td>
<td>111.561</td>
<td>100.541</td>
<td>97.265</td>
<td>113.298</td>
<td>103.7</td>
<td>8.2</td>
</tr>
<tr>
<td>120</td>
<td>120.526</td>
<td>113.564</td>
<td>99.650</td>
<td>130.560</td>
<td>123.840</td>
<td>117.6</td>
<td>11.8</td>
</tr>
<tr>
<td>130</td>
<td>121.121</td>
<td>110.564</td>
<td>114.251</td>
<td>121.541</td>
<td>135.491</td>
<td>120.6</td>
<td>9.5</td>
</tr>
<tr>
<td>140</td>
<td>145.506</td>
<td>135.654</td>
<td>119.262</td>
<td>129.657</td>
<td>131.809</td>
<td>132.4</td>
<td>9.5</td>
</tr>
<tr>
<td>150</td>
<td>145.690</td>
<td>167.800</td>
<td>170.461</td>
<td>131.569</td>
<td>154.210</td>
<td>153.9</td>
<td>16.1</td>
</tr>
</tbody>
</table>
Figure 4.6: Geometric Vergence (100 mm baseline) Mean Values and Error Bars
4.2.2 250 mm Baseline

The final experiment is for a baseline of 250 mm, using the geometric controller. The results are shown in table 4.4 and figure 4.7. The mean and standard deviation values for all distances are acceptable, while not quite as accurate as the gradual controller for the same baseline. Figure 4.8 shows the mean and error values for this experiment.

Table 4.4: Geometric Vergence (250 mm baseline) Experimental Results

<table>
<thead>
<tr>
<th>cm</th>
<th>Sample 1</th>
<th>Sample 2</th>
<th>Sample 3</th>
<th>Sample 4</th>
<th>Sample 5</th>
<th>Mean</th>
<th>Std. Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>16.935</td>
<td>16.847</td>
<td>14.971</td>
<td>15.577</td>
<td>14.412</td>
<td>15.7</td>
<td>1.1</td>
</tr>
<tr>
<td>30</td>
<td>30.176</td>
<td>29.989</td>
<td>32.329</td>
<td>30.314</td>
<td>32.732</td>
<td>31.1</td>
<td>1.3</td>
</tr>
<tr>
<td>35</td>
<td>36.247</td>
<td>34.076</td>
<td>36.914</td>
<td>35.877</td>
<td>35.062</td>
<td>35.6</td>
<td>1.1</td>
</tr>
<tr>
<td>40</td>
<td>39.404</td>
<td>40.639</td>
<td>38.286</td>
<td>41.089</td>
<td>40.101</td>
<td>39.9</td>
<td>1.1</td>
</tr>
<tr>
<td>45</td>
<td>43.382</td>
<td>43.229</td>
<td>45.919</td>
<td>45.983</td>
<td>44.995</td>
<td>44.7</td>
<td>1.3</td>
</tr>
<tr>
<td>50</td>
<td>52.100</td>
<td>49.718</td>
<td>51.220</td>
<td>51.157</td>
<td>50.439</td>
<td>50.9</td>
<td>0.9</td>
</tr>
<tr>
<td>60</td>
<td>56.375</td>
<td>58.244</td>
<td>59.852</td>
<td>61.468</td>
<td>62.131</td>
<td>59.6</td>
<td>2.4</td>
</tr>
<tr>
<td>70</td>
<td>70.869</td>
<td>66.500</td>
<td>72.514</td>
<td>69.046</td>
<td>70.649</td>
<td>69.9</td>
<td>2.3</td>
</tr>
<tr>
<td>80</td>
<td>78.838</td>
<td>77.714</td>
<td>76.546</td>
<td>80.915</td>
<td>82.019</td>
<td>79.2</td>
<td>2.3</td>
</tr>
<tr>
<td>90</td>
<td>94.482</td>
<td>87.077</td>
<td>85.814</td>
<td>90.912</td>
<td>91.204</td>
<td>89.9</td>
<td>3.5</td>
</tr>
<tr>
<td>100</td>
<td>88.513</td>
<td>87.432</td>
<td>99.593</td>
<td>98.175</td>
<td>101.430</td>
<td>95.0</td>
<td>6.6</td>
</tr>
<tr>
<td>110</td>
<td>110.547</td>
<td>93.085</td>
<td>113.886</td>
<td>109.908</td>
<td>111.705</td>
<td>107.8</td>
<td>8.4</td>
</tr>
<tr>
<td>120</td>
<td>109.527</td>
<td>107.414</td>
<td>123.907</td>
<td>119.299</td>
<td>121.421</td>
<td>116.3</td>
<td>7.4</td>
</tr>
<tr>
<td>130</td>
<td>128.418</td>
<td>134.835</td>
<td>130.362</td>
<td>130.632</td>
<td>129.864</td>
<td>130.8</td>
<td>2.4</td>
</tr>
<tr>
<td>140</td>
<td>130.986</td>
<td>135.201</td>
<td>136.067</td>
<td>135.781</td>
<td>142.011</td>
<td>136.0</td>
<td>3.9</td>
</tr>
<tr>
<td>150</td>
<td>130.286</td>
<td>150.442</td>
<td>149.018</td>
<td>150.018</td>
<td>149.328</td>
<td>145.8</td>
<td>8.7</td>
</tr>
</tbody>
</table>
Figure 4.7: Geometric Vergence (250 mm baseline) X Y Scatter Plot for Five Samples at Eighteen Distances

Figure 4.8: Geometric Vergence (250 mm baseline) Mean Values and Error Bars
Chapter 5
Discussion

The two vergence methods presented were each expected to have different advantages and disadvantages. In this chapter, the results of experimentation will be discussed in relation to the expected results. The gradual vergence controller was expected to provide more accurate results, while the geometric vergence controller was expected to be faster.

5.1 Accuracy

The results in the previous chapter demonstrated a high level of accuracy from the depth calculation when the baseline was at the wider setting of 250 mm, for both vergence controllers. A simple accuracy metric has been formed as per equation 5.1

\[
\text{accuracy} = \frac{\sum_{i}^{n} |\text{mean}_i - \text{actual}_i|}{N}
\]

Where \( i \) represents each individual actual distance up to \( n \). \( \text{mean} \) is the sample mean for a given \( i \), \( \text{actual} \) is the real distance at a given \( i \) and \( N \) is the total number of distances measured.

Figure 5.1 displays the above metric for each of the four experiments.

An unexpected result is the difference between the two 100mm baseline experiments. The gradual vergence controller generated much less accurate results than the geometric controller. This is perhaps owing to the physical limitations of the servos. Whereby, the gradual controller attempts to converge to the OoI in small steps. As the servos get closer to the point of convergence, the changes in required vergence become progressively
smaller. However, such small changes will be affected by the loss of floating point precision as the next target value of the servo is converted to the two \texttt{unsigned char} bytes. The Geometric controller does not suffer this problem (the $\Delta$ vergence problem), as the entire angle of required convergence is estimated in one cycle, when the necessary change in vergence angle is not as small.

The $\Delta$ vergence problem has less effect when the baseline is greater. Figure 5.1 demonstrates this by showing the required changes in vergence angle for two different baseline IPDs. \textit{Left}(1) and \textit{Right}(1) represent the two cameras of one configuration, while \textit{Left}(2) and \textit{Right}(2) represent the two cameras of another, wider IPD configuration. This important feature of this figure is the difference in the angles of vergence required by each of the two IPD configurations as they rotate the point-of-gaze from object \textit{A} to object \textit{B}. Note that for the wider IPD angles ($\theta_l(2)$ and $\theta_r(2)$), the required change in vergence angle is greater than for the narrower IPD ($\theta_l(1)$ and $\theta_r(1)$). This also explains why the accuracy of the experiments where the baseline is 250 mm is higher than the corresponding experiments where the baseline is 100 mm. This was an expected result.

Image resolution also has an effect on the accuracy of experimentation. If the resolution of the two captured frames were greater, with it would come an increased precision in the location of the OoI. However, the speed of the system would suffer, as explained in the next section (5.2).
Figure 5.2: Delta Vergence Angles Between Two Objects for Two baseline IPDs
In summary, both vergence methods calculated depth in the same way but it was expected that the gradual vergence would provide better results in terms of accuracy, due to the constant stream of packets sent to the servos until the object of interest was the point of gaze. The geometric vergence controller however, calculated the required rotation angle for each servo after each frame. In the case of the 250mm baseline experiments, the results were as expected. The 100mm baseline, produced an unexpected result where the geometric vergence controller was in fact better than the gradual controller.

5.2 Speed

The low resolution of the captured frames enabled the system to operate in near real-time but prevented the face detection algorithm from identifying faces more than two metres away (approximately). The gradual vergence controller was noticeably and expectedly slower as it had to process a captured frame after each change in vergence. For this reason, the naive implementation of the gradual controller makes it a non-viable algorithm for use in a practical implementation of object tracking with stereo vision. The geometric vergence controller on the other hand needed only to process one captured frame before determining the servo targets. As such, it is very fast (and sufficiently accurate).
6.1 Possible Future Work

The experimentation with this system is not exhaustive. To better compare the geometric vergence method with other vergence control algorithms, these algorithms could be implemented with the same hardware set-up.

The rate at which the Viola-Jones object detection framework is able to search each image for the object of interest could be improved with the inclusion of a restricted search region. The reduction in processing time would lead to an improved rate of capture. The size of the search region could be determined using position and velocity vectors of the OoI. The modular nature of the source code means the object detection method could be replaced altogether. Simpler detection methods would also contribute to a reduced processing time and improve the likelihood of real-time object tracking.

Presently, the system has no method for determining the “best time” to capture the next frame. This means frames can be captured while the vergence servos are in motion. The result is an incorrect required vergence angle calculation for the iteration of the vergence controller algorithm. If frames were only captured after the previous vergence command had been executed, the chance of over-compensated vergence angle requirements will be eliminated. In the current state, the geometric vergence system has two methods for waiting between frame captures. Either by user control (on-command) or by a specified “sleep” period that is long enough to account for all possible durations between the the initial sending of the command and the completion of the servo rotation.
The use of cameras with different lens properties (e.g., the FoV) may provide interesting results. In this work, only cameras with 75° horizontal FoV angles have been used. Also, the geometric vergence controller could be adapted for vertical tilt control and perhaps different methods for integrating vergence and pan motion could be investigated so as to develop a robotic head that can track objects in real-time, with minimal servo interfacing.

6.2 Conclusions

To conclude, this research has introduced a vergence controller that uses trigonometric properties for a purpose-built stereo vision head that is capable of tracking and rotating the point of gaze toward an object of interest. Sufficiently accurate depth measurements can be calculated through geometric triangulation. The system presented is a proof-of-concept which provides a platform for further expansion on the ideas presented.

The objectives of the project have, for the most part, been met with success. A hardware-software combined system has been developed, the system uses some of the properties of biological stereopsis, i.e., pan, tilt and vergence mechanisms, a method for object detection has been implemented, i.e., Viola-Jones frontal-face detection and the system has been tested, producing some positive results.
Appendix A. Serial Servo Controller Class (in C++)

source/SerialServoController.h

```c
/**
 * File: SerialServoController.h
 * Author: Jeremy M. Willense, 2012
 * Description: SerialServoController class - interfaces with the Pololu Micro-Maestro 6-Servo Controller, implementing the compact protocol as described at http://www.pololu.com/docs/0J40/5.e
 */

#ifndef SERIALSERVOCONTROLLER_H
#define SERIALSERVOCONTROLLER_H

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>

// Baud rate
#define BAUDRATE B38400

// Unix device files (up to three file open attempts)
#define DEVICE  "/dev/ttyACM0"
#define DEVICE_2  "/dev/ttyACM1"
#define DEVICE_3  "/dev/ttyACM2"

// Servo positions in microseconds
#define DEFAULT_SERVO_POSITION 1496*4
#define MIN_SERVO_POSITION 992*4
#define MAX_SERVO_POSITION 2000*4
#define UNLIMITED_SERVO_SPEED 0
#define MIN_SERVO_SPEED 1
#define MAX_SERVO_SPEED 255

/*
 * Servo channel numbers
 */
enum {
    PAN_SERVO = 0,
    TILT_SERVO = 1,
    LEFT_EYE_SERVO = 2,
    RIGHT_EYE_SERVO = 3,
    NUMBER_OF_SERVOS = 4
};

/*
 * Servo specific information
 */
```
typedef struct Servo {
  unsigned char channelNumber;
  int position;
  int speed;
  int acceleration;
} Servo;

class SerialServoController {
private:
  // Device file descriptor
  int fileDescriptor;
  // Terminal I/O
  struct termios oldTIO, newTIO;
  // Servos controlled by the Serial Servo Controller
  struct Servo servo[NUMBER_OF_SERVOS];

public:
  // Constructor/Destructor
  SerialServoController();
  ~SerialServoController();
  // Protocol methods
  void resetServos();
  void shutdownServos();
  void setTarget(unsigned char channel, int target);
  void setTargetMini(unsigned char channel, unsigned char target);
  void setSpeed(unsigned char channel, int speed);
  void setAcceleration(unsigned char channel, int acceleration);
  int getPositionFromDevice(unsigned char channel);
  // Get servo specific data
  int getPosition(unsigned char channel);
  int getSpeed(unsigned char channel);
  int getAcceleration(unsigned char channel);
  // Static methods for extracting lower/higher 7 bits of an int
  static unsigned char getLower7Bits(int value);
  static unsigned char getHigher7Bits(int value);
};

#endif // SERIALSERVOCONTROLLER_H
SerialServoController::SerialServoController()
{
    // Open the device file - Makes three attempts with different file names
    fileDescriptor = open(DEVICE, O_RDWR | O_NOCTTY);
    if (fileDescriptor < 0)
    {
        perror(DEVICE);
        fileDescriptor = open(DEVICE_2, O_RDWR | O_NOCTTY);
        if (fileDescriptor < 0)
        {
            perror(DEVICE_2);
            fileDescriptor = open(DEVICE_3, O_RDWR | O_NOCTTY);
            if (fileDescriptor < 0)
            {
                perror(DEVICE_3);
                exit(-1);
            }
        }
    }

    // Save current port settings
    tcgetattr(fileDescriptor, &oldTIO);

    // Clear newTIO bytes to zero values
    bzero(&newTIO, sizeof(newTIO));

    // Port settings for the Pololu device
    newTIO.c_iflag = 0;
    newTIO.c_lflag = 0;
    newTIO.c_oflag = 0;
    newTIO.c_cflag = BAUDRATE | CS8 | CLOCAL;
    newTIO.c_cc[VTIME] = 0;
    newTIO.c_cc[VMIN] = 1;

    // Flush and apply settings
    tcflush(fileDescriptor, TCIFLUSH);
    tcsetattr(fileDescriptor, TCSANOW, &newTIO);

    // Set servos to default targets
    resetServos();
}

/* Destructor - Turn all servos off and restore port settings */
SerialServoController::~SerialServoController()
{
    shutdownServos();

    // Restore original port settings
    tcsetattr(fileDescriptor, TCSANOW, &oldTIO);
}

/* resetServos() - Set all servos to the default target (centre) */
void SerialServoController::resetServos()
{
    for (unsigned char i = 0; i < NUMBER_OF_SERVOS; ++i)
    {
    }
// Initialise the servo structure data
int servo[i].channelNumber = i;
servo[i].position = DEFAULT_SERVO_POSITION;
servo[i].speed = UNLIMITED_SERVO_SPEED;
servo[i].acceleration = UNLIMITED_SERVO_SPEED;

// Send initial commands to the controller
setTarget(i, DEFAULT_SERVO_POSITION);
setSpeed(i, UNLIMITED_SERVO_SPEED);
setAcceleration(i, UNLIMITED_SERVO_SPEED);

void SerialServoController::shutdownServos()
{
    for (unsigned char i = 0; i < NUMBER_OF_SERVOS; ++i)
    {
        setTarget(i, 0);
    }
}

/*
** setTarget() - Send a command to a particular servo channel specifying a
target position
*/
void SerialServoController::setTarget(unsigned char channel, int target)
{
    target = target > MAX_SERVO_POSITION ? MAX_SERVO_POSITION : (target < MIN_SERVO_POSITION && target != 0 ? MIN_SERVO_POSITION : target);
    unsigned char command[4];
    command[0] = 0x84;
    command[1] = channel;
    command[2] = getLower7Bits(target);
    command[3] = getHigher7Bits(target);

    if (write(fileDescriptor, command, 4) != 4)
    {
        printf("Error writing to the device.\n");
        return;
    }

    servo[channel].position = target;
}

/*
** setTargetMini - Use the Mini Protocol to set the position of a servo
*/
void SerialServoController::setTargetMini(unsigned char channel, unsigned char target)
{
    unsigned char command[3];
    command[0] = 0xFF;
    command[1] = channel;
    command[2] = target;

    // Send the command
if (write(fileDescriptor, command, 3) != 3)
{
    printf("Error writing to the device.\n");
    return;
}

servo[channel].position = getPositionFromDevice(target);

/*************************************************
* setSpeed() - Send a command to a particular servo specifying a speed
* limit (1-255), 0 is unlimited
*/
void SerialServoController::setSpeed(unsigned char channel, int speed)
{
    // Compact protocol: 0x87, channel number, speed low bits, speed high bits
    unsigned char command[4];
    command[0] = 0x87;
    command[1] = channel;
    command[2] = getLower7Bits(speed);
    command[3] = getHigher7Bits(speed);

    // Send the command
    if (write(fileDescriptor, command, 4) != 4)
    {
        printf("Error writing to the device.\n");
        return;
    }

    servo[channel].speed = speed;
}

/*************************************************
* setAcceleration() - Send a command to a particular servo specifying an
* acceleration rate
*/
void SerialServoController::setAcceleration(unsigned char channel, int acceleration)
{
    // Compact protocol: 0x89, channel number, acceleration low bits, acceleration high bits
    unsigned char command[4];
    command[0] = 0x89;
    command[1] = channel;
    command[2] = getLower7Bits(acceleration);
    command[3] = getHigher7Bits(acceleration);

    // Send the command
    if (write(fileDescriptor, command, 4) != 4)
    {
        printf("Error writing to the device.\n");
        return;
    }

    servo[channel].acceleration = acceleration;
}

int SerialServoController::getPositionFromDevice(unsigned char channel)
{
    unsigned char command[2];
    command[0] = 0x90;
command[1] = channel;

// Send the command
if (write(fileDescriptor, command, 2) != 2)
{
    printf("Error writing to the device.\n");
    return 0;
}

// Read the response
unsigned char buffer[2];
while (read(fileDescriptor, buffer, 2) != 2)
{
    printf("Error reading from the device.\n");
    return 0;
}
return (int)((buffer[1] << 7) | buffer[0]);

int SerialServoController::getPosition(unsigned char channel)
{
    return servo[channel].position;
}

int SerialServoController::getSpeed(unsigned char channel)
{
    return servo[channel].speed;
}

int SerialServoController::getAcceleration(unsigned char channel)
{
    return servo[channel].acceleration;
}

/*
 * getLower7Bits() - Gets the lower 7 bits of a 14 bit value
 */
unsigned char SerialServoController::getLower7Bits(int value)
{
    return (unsigned char)(value & 0x7F);
}

/*
 * getHigher7Bits() - Gets the higher 7 bits of a 14 bit value
 */
unsigned char SerialServoController::getHigher7Bits(int value)
{
    return (unsigned char)((value >> 7) & 0x7F);
}
source/StereoCamera.h

```cpp
#ifndef STEREOCAMERA_H
#define STEREOCAMERA_H

#include "cv.h"
#include "highgui.h"
#include <cstdio>

enum WindowType
{
    LEFT_AND_RIGHT,
    BINOCULAR
};

class StereoCamera
{
protected:
    CvCapture * capture0;
    CvCapture * capture1;
    IplImage * frame0;
    IplImage * frame1;
    bool leftAndRightVisible;
    bool binocularVisible;

public:
    StereoCamera (bool showLeftAndRight = false, bool showBinocular = false);
    ~StereoCamera();
    virtual bool getFrame();

    int width;
    int height;
};

#endif // STEREOCAMERA_H
```

source/StereoCamera.cpp

```cpp
#include "StereoCamera.h"

StereoCamera::StereoCamera(bool showLeftAndRight, bool showBinocular)
{
    leftAndRightVisible = showLeftAndRight;
    binocularVisible = showBinocular;

    capture0 = cvCaptureFromCAM(1);
    //cvSetCaptureProperty(capture0, CV_CAP_PROP_FRAME_WIDTH, 640.0);
    //cvSetCaptureProperty(capture0, CV_CAP_PROP_FRAME_HEIGHT, 480.0);
```
if (!capture0) {
    fprintf(stderr, "ERROR: capture 0 is NULL \n");
    exit(-1);
}

capture1 = cvCaptureFromCAM(2);
//cvSetCaptureProperty(capture1, CV_CAP_PROP_FRAME_WIDTH, 640);
//cvSetCaptureProperty(capture1, CV_CAP_PROP_FRAME_HEIGHT, 480);
width = cvGetCaptureProperty(capture1, CV_CAP_PROP_FRAME_WIDTH);
height = cvGetCaptureProperty(capture1, CV_CAP_PROP_FRAME_HEIGHT);
if (!capture1) {
    fprintf(stderr, "ERROR: capture 1 is NULL \n");
    exit(-1);
}

StereoCamera::~StereoCamera() {
    cvReleaseCapture(&capture0);
    cvReleaseCapture(&capture1);
}

void StereoCamera::setWindowVisible(WindowType window, bool visible) {
    if (window == LEFT_AND_RIGHT) {
        if (visible) {
            leftAndRightVisible = true;
            cvNamedWindow("Left Eye", CV_WINDOW_AUTOSIZE);
            cvNamedWindow("Right Eye", CV_WINDOW_AUTOSIZE);
        } else {
            leftAndRightVisible = false;
            cvDestroyWindow("Left Eye");
            cvDestroyWindow("Right Eye");
        }
    } else if (window == BINOCULAR) {
        if (visible) {
            binocularVisible = true;
            cvNamedWindow("Binocular Image", CV_WINDOW_AUTOSIZE);
        } else {
            binocularVisible = false;
            cvDestroyWindow("Binocular Image");
        }
    } else {
        // Do nothing
    }
}
bool StereoCamera::getFrame()
{
    // Get one frame
    frame0 = cvQueryFrame(capture0);
    frame1 = cvQueryFrame(capture1);
    if (!frame0)
    {
        fprintf(stderr, "ERROR: frame 0 is null...\n");
        return false;
    }
    if (!frame1)
    {
        fprintf(stderr, "ERROR: frame 1 is null...\n");
        return false;
    }
    if (leftAndRightVisible)
    {
        cvShowImage("Left Eye", frame0);
        cvShowImage("Right Eye", frame1);
    }
    return true;
}
Appendix C. Face Detection Class (in C++)

source/FaceDetection.h

```cpp
#ifndef FACEDETECTION_H
#define FACEDETECTION_H

#include "StereoCamera.h"
#include <stdlib.h>
#include <string.h>
#include <cmath.h>
#include <math.h>
#include <assert.h>
#include <float.h>
#include <limits.h>
#include <time.h>
#include <ctype.h>

class FaceDetection : public StereoCamera
{
  public:
    FaceDetection(bool showLeftAndRight = false, bool showBinocular = false) : StereoCamera(showLeftAndRight, showBinocular)
    {
      storage0 = 0;
      storage1 = 0;
      cascade = 0;
      cascade = (CvHaarClassifierCascade*)cvLoad("haarcascade_frontalface_alt.xml", 0, 0, 0);
      storage0 = cvCreateMemStorage(0);
      storage1 = cvCreateMemStorage(0);
      feed0 = cvQueryFrame(capture0);
      feed1 = cvQueryFrame(capture1);
      frame0 = cvCreateImage(cvSize(feed0->width, feed0->height), IPL_DEPTH_8U, feed0->nChannels);
      frame1 = cvCreateImage(cvSize(feed1->width, feed1->height), IPL_DEPTH_8U, feed1->nChannels);
      frame0Copy = cvCreateImage(cvSize(frame0->width, frame0->height), IPL_DEPTH_8U, frame0->nChannels);
      frame1Copy = cvCreateImage(cvSize(frame1->width, frame1->height), IPL_DEPTH_8U, frame1->nChannels);
    }

  ~FaceDetection()
  {
  }

  "FaceDetection();

  bool getFrame()
  {
    "Get one frame
    feed0 = cvQueryFrame(capture0);
    feed1 = cvQueryFrame(capture1);
    frame0 = cvCloneImage(feed0);
    frame1 = cvCloneImage(feed1);

  return true;
```

63
if (!frame0)
{
    fprintf(stderr, "ERROR: frame 0 is null...\n");
    return false;
}
if (frame0->origin == IPL_ORIGIN_TL)
{
    cvCopy(frame0, frame0Copy, 0);
} else
{
    cvFlip(frame0, frame0Copy, 0);
}
if (!frame1)
{
    fprintf(stderr, "ERROR: frame 1 is null...\n");
    return false;
}
if (frame1->origin == IPL_ORIGIN_TL)
{
    cvCopy(frame1, frame1Copy, 0);
} else
{
    cvFlip(frame1, frame1Copy, 0);
}
if (leftAndRightVisible)
{
    cvShowImage("Left Eye", frame0);
    cvShowImage("Right Eye", frame1);
}
detectAndDraw(frame0Copy, frame1Copy);
    return true;
}
CvPoint getLeftCentroid()
{
    return leftCentroid;
}
CvPoint getRightCentroid()
{
    return rightCentroid;
}
private:
    // Create memory for calculations
    CvMemStorage* storage0;
    CvMemStorage* storage1;
    // Create a new Haar classifier
    CvHaarClassifierCascade *cascade;
    // Cascade filename
    const char *cascadeName;
    IplImage *frame0Copy;
    IplImage *frame1Copy;
void detectAndDraw(IplImage *i0, IplImage *i1)
{
    int scale = 1;

    // Create a new image based on the input image
    IplImage *temp0 = cvCreateImage(cvSize(i0->width / scale, i0->height / scale), 8, 3);
    IplImage *temp1 = cvCreateImage(cvSize(i1->width / scale, i1->height / scale), 8, 3);

    // Create two points to represent the face locations
    CvPoint pt1, pt2;
    int i;

    // Clear the memory storage which was used before
    cvClearMemStorage(storage0);
    cvClearMemStorage(storage1);

    // Find whether the cascade is loaded, to find the faces. If yes
    // then:
    if(cascade)
    {
        // There can be more than one face in an image. So create a
        // growable sequence of faces.
        // Detect the objects and store them in the sequence
        CvSeq *faces0 = cvHaarDetectObjects(i0, cascade, storage0,
                                             1.1, 2, CV_HAAR_DO_CANNY_PRUNING, cvSize(20, 20));
        CvSeq *faces1 = cvHaarDetectObjects(i1, cascade, storage1,
                                             1.1, 2, CV_HAAR_DO_CANNY_PRUNING, cvSize(20, 20));

        // Loop the number of faces found.
        for (i = 0; i < (faces0 ? faces0->total : 0); i++)
        {
            // Create a new rectangle for drawing the face
            CvRect* r = (CvRect*)cvGetSeqElem(faces0, i);

            // Find the dimensions of the face, and scale it if
            // necessary
            pt1.x = r->x * scale;
            pt2.x = (r->x + r->width) * scale;
            pt1.y = r->y * scale;
            pt2.y = (r->y + r->height) * scale;

            // Draw the rectangle in the input image
            cvRectangle(i0, pt1, pt2, CV_RGB(255, 0, 0), 3, 8, 0);

            // Set the centroid
            leftCentroid.x = pt1.x + ((pt2.x - pt1.x) / 2);
            leftCentroid.y = pt1.y + ((pt2.y - pt1.y) / 2);
        }
    }
}

for (i = 0; i < (faces1 ? faces1->total : 0); i++)
{
    // Create a new rectangle for drawing the face
CvRect* r = (CvRect*)cvGetSeqElem(faces1, i);

// Find the dimensions of the face, and scale it if necessary
pt1.x = r->x * scale;
pt2.x = (r->x + r->width) * scale;
pt1.y = r->y * scale;
pt2.y = (r->y + r->height) * scale;

// Draw the rectangle in the input image
cvRectangle(i1, pt1, pt2, CV_RGB(255,0,0), 3, 8, 0);

// Set the centroid
rightCentroid.x = pt1.x + ((pt2.x - pt1.x) / 2);
rightCentroid.y = pt1.y + ((pt2.y - pt1.y) / 2);
}
}

if (leftAndRightVisible)
{
    cvShowImage("Left Eye", i0);
    cvShowImage("Right Eye", i1);
}

} // Release the temp image created.
cvReleaseImage(&temp0);
cvReleaseImage(&temp1);

} // FACEDETECTION_H
Appendix D. Robotic Head Controller Class (in C++)

source/RoboticHead.h

```c++
#include "cv.h"
#include "highgui.h"
#include <cmath>
#include "FaceDetection.h"
#include "SerialServoController.h"

#define MICROSECONDS_PER_DEGREE 44.8f  // (MAX_SERVO_POSITION - MIN_SERVO_POSITION)/90.0
#define BASELINE 10  // cm

enum {
    LEFT_EYE = 0,
    RIGHT_EYE = 1
};

class RoboticHead {
public:
    RoboticHead() {
        controller = new SerialServoController();
        camera = new FaceDetection(true, false);
        neutralAnglePosition[0] = neutralAnglePosition[1] = DEFAULT_SERVO_POSITION;
        degrees1Prev = degrees2Prev = 0;
    }

    ~RoboticHead() {
        delete controller;
        delete camera;
    }

float getCyclopeanAngle(int eye) {
    // + 2 represents the servo channel offset for the left and right cameras (channels 0 and 1 are pan and tilt)
    float difference = fabs((float)controller->getPosition(eye + 2) - (float)neutralAnglePosition[eye]);
    return difference / MICROSECONDS_PER_DEGREE;
}

int degreesToPosition(float angle) {
    return (int) angle * MICROSECONDS_PER_DEGREE;
}

void startConfiguration() {
    while (1)
```


if (!camera->getFrame())
{
    return;
}

if (!waitForControllerAdjustments())
{
    return;
}

void startGradualVergence()
{
    // Loop indefinitely
    while (1)
    {
        // Get a frame (from each camera)
        if (!camera->getFrame())
        {
            return;
        }

        // Locate the centroid of the object in each camera
        CvPoint left, right;
        left = camera->getLeftCentroid();
        right = camera->getRightCentroid();

        // Adjust the point of gaze of the cameras to the centroid
        if (left.x > (camera->width / 2))
        {
            controller->setTarget(LEFT_EYE_SERVO, controller->
                getPosition(LEFT_EYE_SERVO) - (left.x - (camera->
                    width / 2)));
        }
        else if (left.x < (camera->width / 2))
        {
            controller->setTarget(LEFT_EYE_SERVO, controller->
                getPosition(LEFT_EYE_SERVO) + ((camera->width / 2) -
                    left.x));
        }

        if (right.x > (camera->width / 2))
        {
            controller->setTarget(RIGHT_EYE_SERVO, controller->
                getPosition(RIGHT_EYE_SERVO) - (right.x - (camera->
                    width / 2)));
        }
        else if (right.x < (camera->width / 2))
        {
            controller->setTarget(RIGHT_EYE_SERVO, controller->
                getPosition(RIGHT_EYE_SERVO) + ((camera->width / 2) -
                    right.x));
        }

        if (left.y > (camera->height / 2))
        {
            controller->setTarget(TILT_SERVO, controller->
                getPosition(TILT_SERVO) - (left.y - (camera->height

68
else if (left.y < (camera->height / 2))
{
    controller->setTarget(TILT_SERVO, controller->
    getPosition(TILT_SERVO) + ((camera->height / 2) -
    left.y));
}

int key = cvWaitKey(1) & 255;
if (key == 27 || key == '\n') return;
if (key == ' ')
{
    float baseline = BASELINE; // cm
    float degrees1 = 90.0 - get CyclopeanAngle(LEFT_EYE);
    float degrees2 = 90.0 - get CyclopeanAngle(RIGHT_EYE);
    float theta1 = degrees1 * (CV_PI / 180);
    float theta2 = degrees2 * (CV_PI / 180);
    float tanTheta1 = tan(theta1);
    float tanTheta2 = tan(theta2);

    float depthtan = baseline * ((tanTheta1 * tanTheta2) / (
    tanTheta1 + tanTheta2));

    // float depthsin = ((baseline * sin(theta1) * sin(theta2)) / sin(CV_PI - theta1 - theta2));
    std::cout << " Depth: " << depthtan << "cm\n";
}


void startTriangulationVergence()
{
    // Loop indefinitely
    while (1)
    {
        // Get a frame (from each camera)
        if (!camera->getFrame())
        {
            return;
        }

        // Locate the centroid of the object in each camera
        CvPoint left, right;
        left = camera->getLeftCentroid();
        right = camera->getRightCentroid();

        float baseline = BASELINE; // cm
        float degrees1 = 90.0 - get CyclopeanAngle(LEFT_EYE);
        float degrees2 = 90.0 - get CyclopeanAngle(RIGHT_EYE);
        float theta1 = degrees1 * (CV_PI / 180);
        float theta2 = degrees2 * (CV_PI / 180);
        float tanTheta1 = tan(theta1);
        float tanTheta2 = tan(theta2);

        float depthtan = baseline * ((tanTheta1 * tanTheta2) / (tanTheta1 + tanTheta2));

        // float depthsin = ((baseline * sin(theta1) * sin(theta2)) / sin(CV_PI - theta1 - theta2));
\[ \sin(CV\_PI - \theta_1 - \theta_2); \]

```cpp
// std::cout << depthtan << " " << depthsin << " cm\n";
```

```cpp
float oppositeSide = depthtan * tan(37.5 * (CV\_PI / 180));
```

```cpp
float halfWidth = camera->width / 2;
```

```cpp
// if in the right half of the image, subtract 1/2 width of image size from the offset
```

```cpp
float leftOffset = left.x < halfWidth ? halfWidth - left.x : halfWidth - (camera->width - left.x);
```

```cpp
float rightOffset = right.x < halfWidth ? halfWidth - right.x : halfWidth - (camera->width - right.x);
```

```cpp
// Scale offset to cm
```

```cpp
leftOffset = oppositeSide * (leftOffset / halfWidth);
rightOffset = oppositeSide * (rightOffset / halfWidth);
```

```cpp
float leftRotateAngle = atan(leftOffset / depthtan) * (180 / CV\_PI);
```

```cpp
float rightRotateAngle = atan(rightOffset / depthtan) * (180 / CV\_PI);
```

```cpp
int leftPositionOffset = degreesToPosition(leftRotateAngle);
int rightPositionOffset = degreesToPosition(rightRotateAngle);
```

```cpp
// Adjust the point of gaze of the cameras to the centroid
```

```cpp
if (left.y > (camera->height / 2)) {
    controller->setTarget(TILT\_SERVO, controller->getPosition(TILT\_SERVO) - (left.y - (camera->height / 2));
}
```

```cpp
else if (left.y < (camera->height / 2)) {
    controller->setTarget(TILT\_SERVO, controller->getPosition(TILT\_SERVO) + ((camera->height / 2) - left.y));
}
```

```cpp
if (degrees1 > degrees2) {
    controller->setTarget(PAN\_SERVO, controller->getPosition(PAN\_SERVO) + ((left.x - (camera->width / 2)));
}
```

```cpp
else if (degrees2 > degrees1) {
    controller->setTarget(PAN\_SERVO, controller->getPosition(PAN\_SERVO) - (((camera->width / 2) - left.x));
}
```

```cpp
int key = cv\_WaitKey(1) & 255;
if (key == 27 || key == 'n') return;
```

```cpp
if (key == ' ') {
    if (left.x > (camera->width / 2)) {
        controller->setTarget(LEFT\_EYE\_SERVO, controller->
```
getPosition(LEFT_EYE_SERVO) - leftPositionOffset;
}

else if (left.x < (camera->width/2))
{
    controller->setTarget(LEFT_EYE_SERVO, controller->
                          getPosition(LEFT_EYE_SERVO) + leftPositionOffset);
}

if (right.x > (camera->width/2))
{
    controller->setTarget(RIGHT_EYE_SERVO, controller->
                          getPosition(RIGHT_EYE_SERVO) -
                          rightPositionOffset);
}

else if (right.x < (camera->width/2))
{
    controller->setTarget(RIGHT_EYE_SERVO, controller->
                          getPosition(RIGHT_EYE_SERVO) +
                          rightPositionOffset);
}

float baseline = BASELINE; // cm
float degrees1 = 90.0 - getCyclopeanAngle(LEFT_EYE);
float degrees2 = 90.0 - getCyclopeanAngle(RIGHT_EYE);
float theta1 = degrees1 * (CV_PI / 180);
float theta2 = degrees2 * (CV_PI / 180);
float tanTheta1 = tan(theta1);
float tanTheta2 = tan(theta2);

float depthTan = baseline * (tanTheta1 * tanTheta2) /
                (tanTheta1 + tanTheta2);
std::cout << "Depth: " << depthTan << "cm\n";

degrees1Prev = degrees1;
degrees2Prev = degrees2;
std::cout << degrees1 << ", " << degrees2 << std::endl;
}

private:
    SerialServoController *controller;
    FaceDetection *camera;

    // The locations of the neutral cyclopean angle (0 degrees) of each
    // eye in microseconds
    int neutralAnglePosition[2];

    // Used to calculate the degree from the current position and
    // neutral position of each eye
    int microsecondsPerDegree;

    float degrees1Prev;
    float degrees2Prev;

    // A wrapper for cvWaitKey() and key press processing
    bool waitForControllerAdjustments(int delay)
    {
        // Check for a key press and remove higher bits using AND
        // operator
        int key = cvWaitKey(delay) & 255;
        }
// If key is 'escape' or 'enter'
if (key == 27 || key == '\n')
{
    std::cout << "Manual Adjustments Complete\n";
    return false;
}
else if (key == ',')
{
    controller->setTarget(LEFT_EYE_SERVO, MIN_SERVO_POSITION);
}
else if (key == '.')
{
    controller->setTarget(LEFT_EYE_SERVO, MAX_SERVO_POSITION);
}
else if (key == 'q')
{
    controller->setTarget(LEFT_EYE_SERVO, controller->getPosition(LEFT_EYE_SERVO) - 0x10);
}
else if (key == 'w')
{
    controller->setTarget(LEFT_EYE_SERVO, controller->getPosition(LEFT_EYE_SERVO) + 0x10);
}
else if (key == 'a')
{
    controller->setTarget(RIGHT_EYE_SERVO, controller->getPosition(RIGHT_EYE_SERVO) - 0x10);
}
else if (key == 's')
{
    controller->setTarget(RIGHT_EYE_SERVO, controller->getPosition(RIGHT_EYE_SERVO) + 0x10);
}
else if (key == 'r')
{
    controller->resetServos();
}
else if (key == '8')
{
    controller->setTarget(TILT_SERVO, controller->getPosition(TILT_SERVO) + 0x10);
}
else if (key == '2') // down arrow
{
    controller->setTarget(TILT_SERVO, controller->getPosition(TILT_SERVO) - 0x10);
}
else if (key == '4') // left arrow
{
    controller->setTarget(PAN_SERVO, controller->getPosition(PAN_SERVO) - 0x10);
}
else if (key == '6') // right arrow
{
    controller->setTarget(PAN_SERVO, controller->getPosition(PAN_SERVO) + 0x10);
}
return true;
});
Bibliography


